USENIX

MACH SYMPOSIUM

AUTUMN 1991

USENIX

SYMPOSIUM PROCEEDINGS

Mach

November 20 - 22, 1991
Monterey, California

For additional copies of these proceedings contact
USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 U.S.A.

The price is $24 for members and $28 for nonmembers.

Outside the U.S.A and Canada, please add
$14 per copy for postage (via air printed matter).

Past USENIX Mach Proceedings (price: member/nonmember)

Mach Workshop October 1990 Burlington, VT $17/20

Outside the U.S.A. and Canada, please add
$9 per copy for postage (via air printed matter).

Proceedings of the USENIX Mach Symposium

USENIX ASSOCIATION

November 20–22, 1991
Monterey, California, U.S.A.

Program and Table of Contents

Mach Symposium
November 21-22, 1991

Thursday, November 21

Friday, November 22

MACH INTERFACES **9:00 - 10:30**

CHANGES TO KERNEL MEMORY MANAGEMENT **11:00 - 12:30**

REAL TIME, RELIABILTY, COMPARISON **2:00 - 3:30**

Alternate Paper **4:00 - 4:20**

Program Committee

Alan Langerman, Chair
Encore Computer Corporation

Larry Allen
Open Software Foundation

Nawaf Bitar
Hewlett-Packard Company

Susan LoVerso
Encore Computer Corporation

Melinda Shore
Cornell University

Michael Young
Transarc

A Fast Mach Network IPC Implementation

Joseph S. Barrera III

jsb@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue Pittsburgh, PA 15213

Abstract

This paper describes an implementation of network Mach IPC optimized for clusters of processors connected by a fast network, such as workstations connected by an Ethernet or processors in a non-shared memory multiprocessor. This work contrasts with earlier work, such as the netmsg server, which has emphasized connectivity (by using robust and widely available protocols such as TCP/IP) and configurability (with an entirely user-state implementation) at the expense of performance.

The issues addressed by this work are support for low latency delivery of small and large messages, support for port capabilities and reference counting, and integration with the existing local Mach IPC implementation. Low latency for small messages requires careful buffer and control flow management; this work is compared with other fast RPC work described in the literature. Low latency for large messages, particularly for faster networks, requires an avoidance of copying, which can be achieved through virtual memory support; the modifications that were necessary to make Mach's virtual memory support inexpensive enough to be useful for this purpose are described. The distributed implementation of port capabilities, port reference counts, and port migration is discussed, and compared with that in the netmsg server. Finally, performance data is presented to quantify the speedup achieved with the described implementation.

1 Introduction

Mach IPC has traditionally been extended over the network by use of the netmsg server [Sansom 88, Julin & Sansom 89], a user-level server which uses general purpose protocols such as TCP/IP. While this approach has connectivity and configurability advantages, it has a serious performance disadvantage. In particular, network Mach RPCs are three to five times slower than network RPCs in other systems on comparable hardware.

An effort currently underway to implement Mach abstractions on a non-shared memory multiprocessor yielded a requirement for faster network Mach IPC. Non-shared memory multiprocessors (such as the Intel iPSC/860 and its successors) have interconnects with high throughput and very low latency; it is inappropriate to burden such fast interconnects with a slow IPC implementation. Fast network IPC is particularly important for such machines since much more IPC on such systems will be remote.

This research was supported by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDS972-90-C-0035.

2 Issues and Implementation

The following sections describe various important issues in constructing a network IPC implementation, and how the described implementation addressed them. These issues include support for low latency delivery of small and large messages, support for port capabilities and reference counting, and integration with the existing local Mach IPC implementation. [Draves 90]

2.1 Message sizes and latency

There are two important cases to optimize in Mach IPC: small messages (less than 128 bytes), and large messages (carrying one or two pages of out-of-line data). Small messages are used for most requests and many replies, whereas large messages are used for transferring data, including file and device access and paging traffic. Buffering in operating system servers and emulators is responsible for the lack of intermediate sized messages. For example, the BSD Unix server translates a read system call into a read from a mapped file area; the read thus either generates no message, if the page containing the read data is resident, or a small request followed by a large reply if the page needs to be faulted in.

For small messages, software latency is the dominant cost. Since there is little data, network throughput is irrelevant. Network latency (including the software cost of setting up the send and the receive) can range from tens to hundreds of microseconds, but it requires a well-optimized IPC system for such costs to be noticeable.

For large messages, network throughput and data-dependent software costs become important. The most common data-dependent software cost is the cost of copying the data between buffers. Limiting the number of copies to one on send and one on receive is moderately straightforward; eliminating all copies is significantly more difficult. One difficulty is that some network interfaces can only send from or receive into special device memory; also, some interfaces do not scatter/gather, and thus may require a copy to add or remove headers.

Although the cost of copying data may not affect asymptotic throughput, it does increase the latency of one or two page messages; since such messages are common, copying is worth avoiding. Copying can fail to affect asymptotic throughput when the copying of one packet can be done in parallel with the network transmission of another; this is feasible with Ethernet since memory-to-memory copying is typically several times faster than Ethernet. When transferring a small number of packets, however, there is less chance for such overlap. Faster networks (with speeds much closer to memory access speeds) also increase the significance of copying.

2.2 Small messages

Small message transfers were optimized by borrowing many of the techniques that have been explored in systems such as Firefly [Schroeder & Burrows 89], Amoeba [van Renesse et al. 88], Sprite [Ousterhout et al. 88], and V [Cheriton & Zwaenepoel 83]. For example, context switches are avoided by having the interrupt thread do as much work as it can and having the thread receiving the message do the rest. A context switch on the sending side of an RPC is avoided by not switching to the idle thread when there are no runnable processes; instead, the sending thread spins, waiting for the reply message (or for another thread to become runnable).

2.3 Large messages

Beyond the optimizations required for small messages, optimizing large messages requires that the data size dependent costs be minimized. As discussed above, the primary data dependent software cost is the cost of copying data, and this cost is particularly significant both for latency and for networks that run close to memory speed. The following sections examine two methods for avoiding copying.

2.3.1 Avoiding copies via shared buffers

One method for avoiding copies, used for example by Firefly RPC, is the use of non-pageable buffers shared between user tasks and the network driver. Users construct messages in a shared buffer; the driver then sends directly from the appropriate buffer. Beyond avoiding both mapping and copying operations, this scheme has the advantage of working with network devices that can only view a subset of the physical memory.

The shared buffer area approach has a number of disadvantages, however. First, a single shared buffer area offers no protection against tasks interfering with each other's message operations. Such protection requires a separate buffer area for each sending task; however, this divides the device space into small pieces, which may artificially limit (in a device-dependent way) the maximum message size. Furthermore, separate buffer areas for each task do not protect the driver from malicious users, particularly if the device does not do scatter/gather and thus must create headers in the shared area.

There are also semantic problems with the shared buffer area approach. Most IPC systems offer by-copy semantics for message sending: once the send completes, modifying the data in the sent buffer will not change the data seen by the receiver. In contrast, when data is sent via a shared buffer, the sender must wait for the driver to indicate that it is done with the buffer. Furthermore, the fact that the shared buffer is a limited resource, and that most of the user's data does not live in the shared buffer, makes it probable that the user task will need to copy its data into or out of the shared buffer upon sending or receiving. Shared buffers may therefore only shift the need for copying onto the user task instead of truly eliminating it.

2.3.2 Avoiding copies via virtual memory mapping

The above-mentioned problems with the shared buffer approach, combined with the difficulty of integrating shared buffers with Mach IPC semantics, led to the decision to avoid copying by using virtual memory operations. This approach has precedent in the Mach system, as local Mach IPC uses copy-on-write to avoid copying data until either the sender or receiver attempts to modify the data.

Copying is avoided upon sending by mapping user data into the kernel address space, faulting in any non-resident pages and marking the pages unpageable. The network driver then uses the kernel mapping of the data. When data is received, it is received into kernel buffers which are then mapped into the receiver's address space.

Unfortunately, the existing Mach methods for manipulating out-of-line data were ill-suited for network IPC, and thus the cost of using the virtual memory system to avoid a copy was several times the cost of the avoided copy. The primary problem is that the Mach virtual memory data structures used to represent copied data are optimized for long term copy-on-write sharing of data, typified by address space copies. In particular, out-of-line data in a message is represented by a *copy object*, which is a complicated data structure

which preserves copying and sharing relationships. When out-of-line data was sent across the network using the original virtual memory data structures, several expensive operations occurred. First, the address in the message was converted into a copy object. The copy object was then mapped into the kernel's address space, and the pages were faulted in and made non-pageable. After the data was sent, the data was unwired and unmapped from the kernel address space.

The key observation about the way data is used when sent remotely is that it is used read-only and then quickly deallocated by the driver, and it only needs to be protected by writing from the user until the driver has completed sending it. This short term sharing of the data suggested a *page list* data structure. Every page in the data to be sent is located, faulted as necessary, and placed in list, which is then handed to the network driver.

There are two methods for preventing the user task from modifying the pages being sent. The first is to prevent the user from returning from the kernel as long as the driver needs the pages. This method works well when the user task is performing a send-receive call and thus will wait anyway for a reply before leaving the kernel. This method also is viable if the device requires a copy (for byte-swapping or limited addressing reasons) and thus will be done with the pages as soon as it has made the copies, or if the network is fast enough that the acknowledgement is likely to arrive by the time the sender has unwound from the routines leading to the driver send routine.

The second method is to protect the pages against writing and let the user return from the send. Instead of using the general and relatively expensive copy-on-write technology, the pages are marked "busy", and protected against writing by calling a pmap routine (which manipulates the page tables directly). When a thread faults on a busy page, it blocks until the page is made not busy; this is a preexisting mechanism in the Mach virtual memory system to indicate that the page is waiting for some event to complete, such as a page to be read from disk. The network driver then marks the page not busy when it is done with it. It does not have to unprotect the page, since the virtual memory system will automatically unprotect the page upon a fault. One reason for not having the network driver unprotect the page is the possibility of the driver accidently granting write permission when it should no longer be allowed.

A separate method for preventing copying is for the user to specify that the kernel should deallocate the sent pages from the user's address space. This option is standard in Mach IPC, and is commonly used by servers which generate but do not cache data, such as pagers or device servers.

These issues are not particular to network IPC; they are applicable whenever the kernel needs to write data to a device, be it a network (used for IPC or standard protocols) disk, or tape. The key feature is the lack of long term sharing, and the fact that the kernel won't write to the data. A separate project has generalized page lists so that they can be used in all such cases. [Black 91]

2.3.3 Avoiding copies on receiving using virtual memory mapping

Copies can be avoided upon receiving as well as sending. When pages of data are received, they are received directly into anonymous pages, unmapped by any user task. These pages are then threaded into page lists, which are then inserted into the Mach message being constructed. Finally, when a user task receives the message, the pages in the page lists are mapped into the task's address space, and the page list pointers in the message are replaced with pointers to the mapped areas. The use of anonymous pages for receiving allows copies

to be avoided without manipulation or examination of the receiver's address space, and without a thread blocked waiting for the message. Other systems, such as V, require that a thread be waiting for a message for the copy to be avoided.

2.4 Integration with local IPC implementation

One of the negative aspects of integrating remote IPC into the in-kernel local IPC system is the possibility of introducing more complexity into an already complex system. Fortunately, the interactions between the local and remote IPC code are limited to two areas: message translation and message queueing.

Local Mach IPC messaging has four stages: copyin, queuing, dequeueing, and copyout. Copyin consists of copying the message buffer from the user's address space into a kernel buffer, and translating ports and out-of-line data into internal kernel representations. The message buffer is then queued on the destination port. When a thread is ready to receive the message, the message is dequeued and copied out, with translation back from kernel to user representations for ports and out-of-line data.

The remote Mach IPC implementation intercepts messages at the queueing stage. When a message is about to be queued on a port, the queueing routine checks whether the port is remote; if it is, it gives the message to the remote IPC system instead of queueing it. This code parallels existing code which checks for messages sent to kernel owned ports such as task, thread, and device ports. Conversely, when the remote IPC implementation receives a message from the network, it inserts it into the local IPC system by calling the queueing routine, as if the message has been sent from a local task.

Similarly, during the translation state, if a message is destined for a remote port, the ports and out-of-line data are translated into over-the-wire representations instead of kernel internal representations. When a message is received from the network, the over-the-wire representations are translated into kernel internal representations before the message is given to the local IPC system.

2.5 Port names, capabilities, and reference counts

The extension of Mach IPC across a network introduces issues such as distributed naming, consistency, and garbage collection. The following sections describe how remote rights are represented, introduce an efficient mechanism for collecting and distributing port usage information, and describe how this mechanism is used to implement port death, port migration, and no senders notifications.

2.5.1 Global port identifiers

When a node sends send rights to a port to another node, it uses a global port identifier to identify the port. This global identifier serves two purposes. First, it allows a node to "merge" port rights, as required by Mach IPC semantics. Second, it provides a method for determining the node to which messages sent to the port should be delivered.

When a task receives send rights for a port that it already holds send rights to, Mach IPC guarantees that the new send rights will have the same name as the old. This merging of port names allows the task to identify two port rights as referring to the same port. The use of global port identifiers allows the kernel to support this merging by allowing it to recognize identical remote send rights.

A fundamental question concerning global identifiers is whether they should encode location information. It is generally cheaper and simpler to locate an object directly from its identifier than to use a separate mechanism to map identifiers to locations; however, some provision must be made for object migration, which invalidates the location information encoded in the object's identifier. Either a new identifier for the object, encoding the new location, must be created and distributed, or the identifier must be entered into a list maintained at every node which lists all identifiers whose location information is incorrect.

Three factors led to the decision to encode port information in the global identifier, and to change identifiers upon port migration. First, port migration is very rare, whereas sending to a port is very common. It is therefore acceptable to complicate port migration in order to reduce the space and time cost of determining port location. Second, the mechanisms required for distributing new identifiers and invalidating old identifiers are already required by other aspects of Mach IPC, such as no-senders notifications. Finally, identifier replacement can be performed with no user impact, since tasks use per-task port names and never see global port identifiers; this decision would have been more difficult if all tasks shared the same port name space.

2.5.2 Proxy ports

A proxy port is the local representative of a port whose receive rights lives on another node; it is created the first time a node receives send rights to the port. A proxy port is treated like a normal port by the local IPC code, and thus automatically maintains per-node usage information about the port. In particular, it maintains local send right counts, which is critical for implementing distributed no senders notification. The global identifier for a port is contained in every proxy for that port. This allows nodes to merge send rights that they have seen before; it also allows the remote IPC code to know where to send a message when the local IPC code attempts to send the message to a proxy.

2.5.3 The periodic token

The periodic token solves several information distribution problems in an inexpensive manner. It allows a single node to broadcast information to all other nodes. It also allows a node to collect information from a large set of nodes. Finally, it allows nodes to know when such information has been seen by all other nodes.

The periodic token is a writable message that is periodically sent on a fixed path through every node in the system. The token passes by each node three times each period. During the first pass, each node writes information into the token. During the second pass, each node reads information from the token. The third pass simply informs each node that all information carried by the token has been seen by every node. To reduce the overhead due to processing incoming token messages, the token pauses for a few seconds in between periods.

The primary advantage of the periodic token is that it batches many small and often self-cancelling messages into one larger message, greatly reducing message handling overhead. It is particularly appropriate for port usage information, since such information is intrinsically self-cancelling (since intermediate reference count values are uninteresting), and does not need to be acted upon any more than once every few seconds.

2.5.4 Detecting no senders

A task can request a no-senders notification message when it attempts to receive from a port for which no task has send rights. This notification allows servers to garbage collect objects which are no longer in use. Detecting no senders is easy in the centralized case; it is significantly harder in the distributed case.

In the centralized case, one data structure describes all current usage of the port, including the number of send rights held by tasks and queued messages in the system. This count is updated by every operation that changes the send right count; it is therefore a simple matter to send a notification to the receiver when this count drops to zero.

In the distributed case, the send right count is distributed across all nodes, and in general no one node has an accurate global count; all a node knows is the local send right count. This lack of knowledge has two causes. First, any holder of either send or receive rights to a port can generate new send rights. If receive rights were required to generate send rights (as is the case for send-once rights), then the holder of receive rights could maintain an accurate count. Second, when a node sends send rights to another node, the sending node cannot assume that the global send count has increased, since the send right might be merged by the receiving task on the receiving node.

A naive approach to implementing no senders detection, in which a pessimistic yet up-to-date send count is maintained at the node holding receive rights, would generate a huge amount of message traffic. In such an approach, a message would be sent to the holder of receive rights whenever a send right was sent from one node to another; another message would be sent by the receiver of the send right if the send right count was not to be incremented. Such a count is pessimistic because it overestimates the global send count. When an accurate count is not possible, a pessimistic count is necessary to prevent incorrect no-senders notifications.

The periodic token can be used for a much more efficient implementation of no senders detection, since the periodic token introduces a fixed, small number of messages into the system. In such an implementation, each node associates a transit count with each proxy; this transit count is incremented when a node sends the send right associated with the proxy, and decremented when it receives such a send right. The purpose of the transit count is to count all send rights held by messages that have been sent and not received. Each period, as the token circulates, each node writes the send count and transit count of its remote send rights into the token.

If the token could visit every node instantaneously, then no senders could be detected by looking for uniformly zero send and transit counts. Unfortunately, as the token passes from one node to the next, the second node can send a send right to the first, receive it back from the first, and then deallocate it; this leaves the already scanned first node with a send right and the second node with a zero transit and send count.

A correct algorithm does exist which requires two passes and a new export flag. This new flag indicates whether any send rights were sent by the node since the previous pass. No senders can now be detected if two passes return zero send and transit counts, and if the export flags indicate that the right was not sent at all between the first and second pass.

2.5.5 Port death

Port death is easily implemented once no senders has been implemented. When a port dies, its death is broadcast using the periodic token. The port and its global identifier can

then be garbage collected as soon as no senders is true. Note that no senders automatically accounts for send rights in transit; simply waiting until the token announcing port death has been seen by every node does not.

2.5.6 Port migration

Like port death, port migration is easily implemented with the help of no senders detection. When a port is migrated from one node to another, a new identifier is allocated with correct location information. This new identifier is broadcast to each node using the periodic token; however, the old identifier is used until every node knows that every other node has seen the new identifier. The danger of using the new identifier too soon is that a node that has not been told about the new identifier will not realize that it refers to the same port as the old identifier, and thus will not correctly merge the port right. When no senders is true for the old identifier, all nodes are free to garbage collect the forwarding information associating the old and new identifiers.

2.6 Reliable delivery and flow control

A network IPC implementation must provide reliable delivery. Unreliable delivery can be caused by packets destroyed or lost by the network, or by packets being dropped by a node because it lacks buffer space. The described implementation was originally designed for reliable networks and thus used a protocol that only handled lack of buffer space through the use of negative acknowledgements. This protocol has since been extended for unreliable networks by adding timeouts while retaining negative acknowledgements.

2.6.1 Protocol for a reliable network

In the original protocol, designed for reliable networks, every packet sent to a node is either positively or negatively acknowledged. A positive acknowledgement allows the sender to send another packet; a negative acknowledgement requires the sender to resend the current packet. A negative acknowledgement is sent only when buffer space was not available when the packet was received, but is available now; it may thus be delayed for however long it takes for the receiver to find more space.

The ability to acknowledge (positively or negatively) every packet requires cooperation from both software and hardware, including limited hardware support for flow control. The software must recognize when it only has one buffer left, and continually reuse that buffer to receive every incoming packet and record nodes that require negative acknowledgements. The hardware must not lose packets that are sent to a node that has not yet rearmed its interconnect with a receive buffer; in practice, since the interconnect does not have infinite buffer space, this means that the interconnect must be willing to delay network sends. The interconnect in the iPSC/860 provides this capability; presumably a token ring could be designed to do so as well. Note that the existence of hardware flow control does not obviate the need for software flow control, as hardware flow control makes the network partially unusable for as long as it needs to hold onto a packet, which can in the worst case cause system wide deadlock.

The primary motivation for using negative acknowledgments is the avoidance of timeouts to detect buffer space overflow. Timeouts are a poor mechanism in this case because of the large variation in time that it takes for a node to find more buffer space, which makes it difficult to select an appropriate timeout value. In some cases, a node just needs to schedule

a thread to perform memory allocation that cannot be performed at interrupt level; in other cases, the node will need to page out to disk before it will have sufficient buffer space.

2.6.2 Adaptation for an unreliable network

To extend this protocol to unreliable networks such as Ethernet, it was necessary to add timeouts and retransmission. Once these mechanisms have been added, negative acknowledgements are no longer necessary; however, we decided to retain negative acknowledgements for two reasons. First, we wanted to use a common protocol for reliable and unreliable networks, and did not want to give up the advantages that negative acknowledgements provide in the reliable case. Second, it is difficult to find a good timeout value when timeouts are used for both packet loss and buffer space depletion.

To combine negative acknowledgements with timeouts, we added two more types of messages: acknowledgement requests and quench requests. When a node sends a packet, it expects an acknowledgement (positive or negative) within two timeout periods. If it does not receive one, it sends an acknowledgement request and waits again. In turn, when a node receives a packet when it has no buffer space, and if it still has no buffer space one timeout period later, it sends a quench message to the sender to prevent a series of acknowledgement request messages. When buffer space becomes available again, the receiver sends a negative acknowledgement, repeated as necessary.

When timeouts are used only for detecting lost packets, it becomes possible to use a much shorter timeout. This is particularly important on inexpensive workstations and on personal computers, which combine a low latency network (Ethernet) with lossy and unreliable network interfaces. In particular, we were faced with Ethernet cards that produced five percent packet loss rates between two machines on the same wire; when combined with a hardware latency around a millisecond, it makes no sense to use timeouts of hundreds of milliseconds, as the BSD TCP implementation does [Leffler et al. 89].

3 Tradeoffs

The primary tradeoff between this implementation and the netmsg server implementation is speed versus generality. This implementation has been primarily designed to support distributed memory multiprocessors; it therefore assumes a fixed number of homogeneous nodes, low network latencies, no independent failures, and no network partitions. In return, it provides fast message delivery with small space costs and timely notifications for events such as port death.

It is possible to obtain both fast IPC with nearby nodes and good connectivity with far-away machines by using the fast IPC implementation among a cluster of nodes and then running a netmsg server on one of the nodes. In this configuration, the netmsg server acts as a gateway. Messages sent within the cluster will be handled entirely by the fast IPC implementation; message sent from a node in the cluster to a machine outside of the cluster will be sent first via fast IPC to the node running the netmsg server, and then to the remote machine by way of the remote machine's netmsg server.

4 Evaluation

The described implementation is considerably faster than the netmsg server, and comparable to the fastest RPC systems described in the literature. A simple Mach RPC using the netmsg server between two i486 PCs running Mach 2.5 (with in-kernel TCP/IP) takes 9.2 milliseconds. In contrast, a simple Mach RPC using the described implementation between the same machines takes 2.5 milliseconds. This represents almost a fourfold improvement over the netmsg server, and is comparable with RPC times for V (2.5 milliseconds) and Sprite (2.8 milliseconds), as measured between sun 3/75 workstations, and for Firefly RPC (2.7 milliseconds) running on a five processor Firefly [Schroeder & Burrows 89].

The described implementation also provides comparable performance to that of proprietary message passing systems on distributed memory multiprocessors. For example, a simple Mach RPC between two 16 Mhz i386s using the iPSC/2 interconnect takes 0.9 milliseconds; this compares to 0.7 milliseconds for NX, Intel's proprietary operating system, running on the same hardware, despite the considerably simpler semantics of NX message passing.

5 Conclusion

A Mach IPC implementation has been described which has been optimized for clusters of processors connected by a fast network. By avoiding the complexities introduced by ill-behaved hosts and networks, adopting optimizations demonstrated in previous fast RPC work, and developing new techniques to avoid copying, the new implementation performs competitively with other RPC systems and considerably faster than the netmsg server implementation, while preserving full Mach IPC semantics. This implementation will serve as a cornerstone for efficient Mach kernel support for distributed memory multiprocessors.

References

[Black 91] Black, D. L. Page Lists and EMMI Work in Progress. Unpublished, June 1991.

[Cheriton & Zwaenepoel 83] Cheriton, D. and Zwaenepoel, W. The Distributed V Kernel and its Performance for Diskless Workstations. In *Proceedings of the Ninth Symposium on Operating System Principles.* ACM, 1983.

[Draves 90] Draves, R. P. A Revised IPC Interface. In *Proceedings of the USENIX Mach Workshop*, pages 101–121, October 1990.

[Julin & Sansom 89] Julin, D. P. and Sansom, R. D. Issues with the Efficient Implementation of Network IPC and RPC in the Mach Environment. Unpublished, September 1989.

[Leffler et al. 89] Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. *The Design and Implementation of the 4.3 BSD Unix Operating System.* Addison-Wesley, Reading, MA, 1989.

[Ousterhout et al. 88] Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., and Welch, B. B. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, February 1988.

[Sansom 88] Sansom, R. D. *Building a Secure Distributed System.* PhD dissertation, Carnegie Mellon University, May 1988.

[Schroeder & Burrows 89] Schroeder, M. D. and Burrows, M. Performance of Firefly RPC. In *Proceedings of the Twelfth Symposium on Operating System Principles.* ACM, 1989.

[van Renesse et al. 88] van Renesse, R., van Staveren, H., and Tanenbaum, A. S. Performance of the World's Fastest Distributed Operating System. *Operating Systems Review*, 22(4):23–34, oct 1988.

Generalized Emulation Services for Mach 3.0
Overview, Experiences and Current Status

Daniel P. Julin *Jonathan J. Chew* *J. Mark Stevenson*
Carnegie Mellon University

Paulo Guedes *Paul Neves** *Paul Roy*
Open Software Foundation

Abstract

This paper reports on an ongoing project to develop a general understanding of the problems encountered when building emulators for various operating systems at the user-level on top of a Mach 3.0 micro-kernel, and to propose a common framework to construct emulations for a wide range of target systems and environments. It presents an overview and discussion of the major techniques and experiments that characterize the design of the proposed system. Some of the relevant aspects include the combination of several independent servers to create a complete system, generic service interfaces relying on the emulation library as an interface translator, the use of object-oriented technology to define standard interfaces and to simplify the implementation of common facilities, moving portions of the system state and processing from the servers into a smart emulation library, and a few general-purpose facilities that simplify the generation of a complete system. A number of practical observations and experiences are also presented, in the context of the development of a prototype for the emulation of UNIX 4.3 BSD.

1. Introduction

Defining and standardizing a powerful micro-kernel base is only the first step in realizing the potential of the architecture proposed with the overall Mach approach. The next step, and perhaps the one richest in design possibilities, is to learn how to construct a wide range of useful higher-level systems on top of this simple kernel. A particular class of such systems is the so-called *emulation systems*, that implement the application programming interface of an existing complete operating system or *target system* with various combinations of user-level components (servers and/or libraries) operating on top of a Mach kernel. The emulation system provides an *operating system environment* for a number of *emulated processes*, such that programs executing in these processes can operate as if they were running under a native implementation of the target system. The main benefits expected from this overall emulation approach, met at various degrees by different system architectures, include increased modularity, portability, flexibility, security and extensibility, as well as simpler development, debugging and maintenance.

With the increasing maturity of the Mach 3.0 micro-kernel, an ever-growing number of pure and applied research groups are now undertaking to design, build and study such emulation systems for a range of targets

---

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDS972-90-C-0035 and in part by the Open Software Foundation (OSF). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, OSF, or the U.S. government.

*Paul Neves is currently with the Swiss Bank Corporation / O'Connor Assoc. L.P., Chicago IL

that span from relatively simple systems like MS-DOS to considerably larger systems such as VMS[8, 1, 11, 9]. All these systems have a few common characteristics: a Mach kernel, one or more servers, and an *emulation library* associated with each emulated process, that intercepts the system calls issued by that process and redirects them to the appropriate emulation services. But beyond these broad similarities, a number of different approaches are taken. Some systems use a single monolithic server that more-or-less reproduces the internal structure of the native implementation of the OS being emulated. Others attempt to decompose the functionality of the single native kernel into several independent servers. Still others follow a mixed approach, with one main central server and a number of auxiliary servers implementing specific portions of functionality that are more easily or usefully separated. Furthermore, some designs attempt to keep all emulation code strictly out of the kernel, while others define various controlled mechanisms to assist emulation by integrating some specialized modules into the protected kernel space.

In general, most efforts to develop emulation systems so far are being pursued in a mostly "ad-hoc" fashion. The precise organization of the interactions between the components of each emulation system is often largely determined by extrapolating from the architecture of the native implementation of the target operating system. Moreover, each individual component is typically designed specifically for the particular system in which it is to be used, and implemented either entirely from scratch or by adapting code from the native OS implementation. However, it appears that a number of basic design and organizational issues, and even the specification of some high-level functions, are in fact quite similar between various emulation systems for different targets. In response, this paper reports on an ongoing effort to take a broader view of the problem of OS emulation in general, and to propose a common framework to minimize the duplication of effort for the development of multiple emulation systems for the widest possible range of target systems and environments. This range may include several different "traditional" operating systems (UNIX, VMS, DOS, etc.), various specialized systems or configurations related to a single main target, or even completely new programming environments.

2. Approach and Overview

The primary focus of this work is the definition and refinement of a general-purpose architecture for operating system emulation. This design is not directly tied to the emulation of any particular operating system, nor does it attempt to imitate the structure of any particular existing operating system implementation. Instead, this work attempts to take an original look at the problems and opportunities presented by a micro-kernel architecture for emulation, and to develop a global understanding of the pertinent design trade-offs. It concentrates on identifying common issues and proposing general solutions or practical mechanisms to address them. The main challenge is to define such a general architecture so that the resulting systems remain practical and competitive when compared with traditional monolithic kernel-based architectures, particularly in the areas of simplicity of implementation, robustness and, of course, performance.

To provide a concrete environment in which to evaluate the design, we are implementing a prototype of a complete emulation system for UNIX 4.3 BSD, often referred to as the "Mach 3.0 multi-server emulation system". However, this particular prototype is not intended to be the major or single final product of this work. Rather, it is used as a vehicle with which to test general ideas about the system organization and to experiment with particular implementations of these ideas. The overall development of the design proceeds through successive refinements of the prototype, such that each consecutive version constitutes a new working emulation environment with better characteristics than the one preceding it, while allowing all design decisions to be reconsidered at each stage in light of the experiences accumulated during this process. The "final" prototype after the design and evaluation are complete might then be used as a basis for further engineering work to construct a production-quality system for BSD emulation, or for other UNIX emulations.

The overall architecture is shown on figure 1. The major elements in this architecture are the kernel, the emulated processes, and a collection of servers.

The kernel is the "pure" Mach kernel, exporting only the basic Mach primitives. It is completely general,

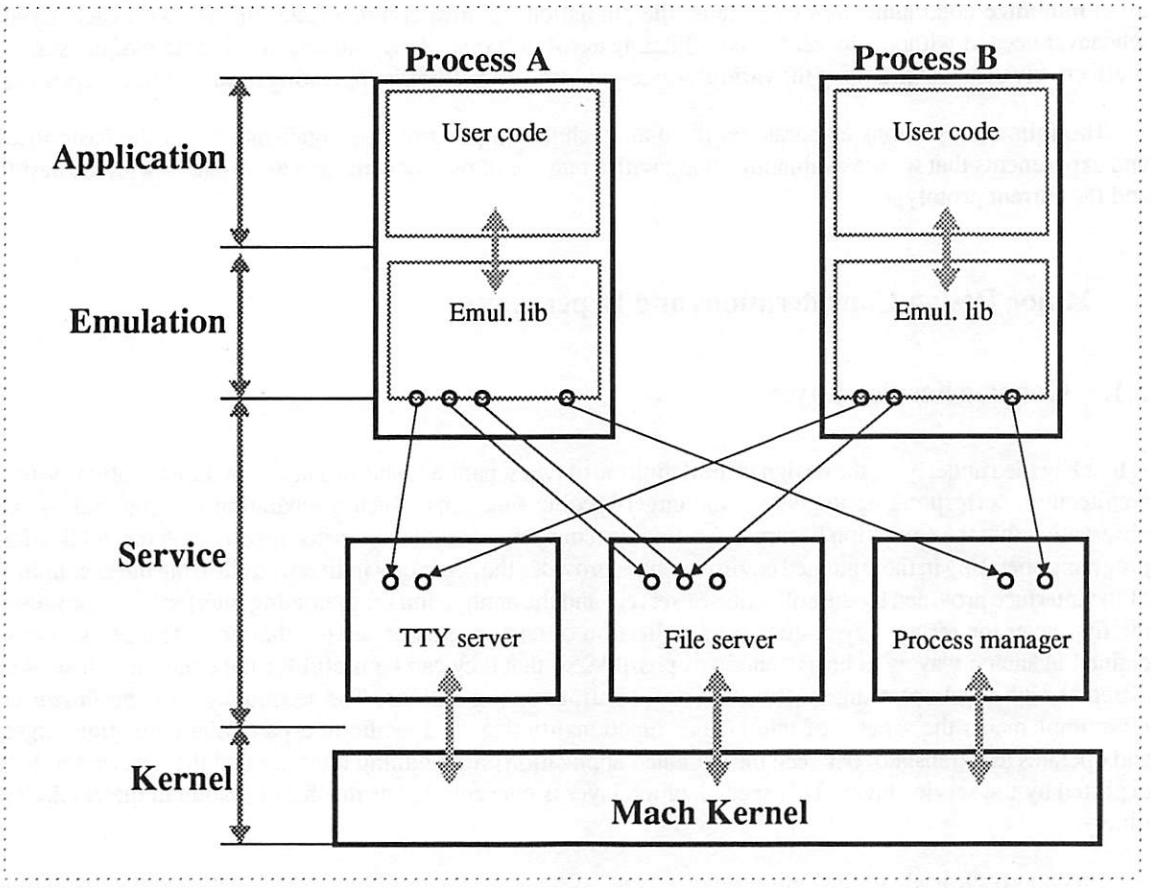


Figure 1: Overall System Architecture

and contains no code specific to any emulation system. This decision is intended to maximize security and fault isolation.

Each emulated process is implemented in a separate Mach task that contains the unmodified user code from various application programs and an emulation library that intercepts and implements system calls issued by instructions embedded in the user code. Although both of these regions of code reside in the same address space, we often refer to the circumstances when a thread is executing user code as "executing in *user space*", and when a thread is executing code in the emulation library as "executing in *emulation space*". The emulation library is relatively large; it manages a considerable portion of the process state and the emulation functions locally.

The rest of the functionality of the emulation system, that is not handled in the emulation library itself, is distributed among a "federation" of servers each operating in a separate Mach task. These servers are independent of each other and can be replaced or re-configured at will, to provide for maximum flexibility. Examples of servers in the BSD prototype include file servers, a process manager, a TTY server, a pipe/socket server, etc.

The primary mode of communication between components in the system is the Mach IPC facility. Every individual UNIX abstraction (files, sockets, pipes, etc.) in every server is represented by a separate Mach port. However, Mach shared memory may also be used in client-server interactions when appropriate: for example, file data is typically mapped directly into the address space of each client for fast I/O access.

Finally, to maximize the potential for building different configurations or collections of servers, as well

as to minimize communication overheads, the emulation libraries communicate directly with each server whenever needed without any central coordinating agent or server. As a consequence, the state of the system is effectively distributed among the various servers and emulation libraries operating in each emulated process.

The following sections elaborate on the main technical aspects of this organization and the basic ideas and experiments that we are evaluating, along with a number of our concerns and experiences with the design and the current prototype.

3. Major Design Considerations and Experiences

3.1. Generic Service Layer

A first key idea underlying the design is the definition of two separate functional layers in the emulation system architecture, corresponding to generic and target-specific functions. Such a separation is suggested by the observation that the emulation library is the *only* system component that interacts directly with the application programs operating in the emulated environment; it provides the necessary indirection allowing the decoupling of the interface provided by the collection of servers and the application programming interface. Accordingly, the first layer, or *service layer*, contains a collection of components or servers that provide basic services, defined in such a way as to be as generic as possible, so that they can be useful for the emulation of several different high-level operating systems and/or for various configurations. The second layer or *specialization layer* implements the aspects of interface or functionality that are specific to a particular emulation target, and operates as a translator between the emulated application programming interface and the generic services exported by the service layer. This specialization layer is concentrated as much as possible in the emulation library.

This approach leads to the definition of a new "system programming interface" between the emulation library and the service layer that is not directly visible to application programs. Its design can thus concentrate on issues of security, performance and most of all flexibility, thereby increasing the potential for making the service layer more generic. Issues of simplicity and ease of use of this interface by programmers, while still important, are nonetheless secondary when compared to this main consideration. Similarly, the modes of interactions between the components inside the service layer can be defined freely to maximize the flexibility, configurability and reusability of this layer, with little concern for the requirements of application programs in each specific target system.

In practice, the service layer can, of course, only be partially generic. Current observations from the design of our prototype indicate that high-level services such as file management, network access and local "pipe-style" interprocess communication, can be expected to be directly useful for a variety of operating system environments. In addition, many ancillary services that are not directly exported at the application program level, such as authentication, location of servers (coarse-grain naming), configuration management, etc., can also be reused in many different system configurations. On the other hand, process management remains largely tied to the semantics of each particular target operating system, mainly because of the complexity of the definitions of groups of emulated processes and the associated rules for access mediation. However, we have started and continue to develop a set of low-level core components that can be assembled and customized to construct different variations of process managers. Finally, terminal management ("TTY") has not been considered for generalization so far; the main difficulties in this area stem from the semantic richness of the line management strategies, together with the multiple user-controllable options (*ioctl()*).

3.2. Standard Object-Oriented System Interfaces

To capitalize on the idea of using generic services, most high-level functions are defined in terms of an object-oriented framework. Each operating system service is represented by one or more abstract *OS objects* or *items*, exporting a well-defined set of operations. Examples from the UNIX domain are files, pipes, sockets, tty's, etc., but in practice, each of those UNIX abstractions may, of course, be represented by a more neutral item, corresponding to a generic service that can be specialized by the emulation library for a given emulated environment. Each server normally implements a large number of similar, but independent items. Note that the various servers and items may themselves be implemented using object-oriented techniques; the word *item* is used to avoid confusion with the actual objects used at the implementation level.

The operations exported by each item are categorized into several independent functional groups which are standard throughout the system. Each such group of functions is represented by a specific "standard" interface. These interfaces are important both to allow a degree of target-system independence at the service level, and to allow for the easy combination and integration of many independent servers or services. The major categories of functions currently defined for our BSD prototype, that appear to be well-suited in both of these respects, are:

access mediation: access to all entities (*items*) in the system is mediated through a standard facility using general-purpose access control lists and a uniform representation for user credentials. The requirements of particular target OS'es can typically be handled with special initialization and interpretation of the standard abstractions. There is a "secure" variation of this system, targeted at the B3 level of security[11].

naming: all entities in the system can be named and accessed through a uniform name space, that also handles garbage-collection and access mediation for item creation. There is a common name resolution protocol to navigate the global name space, locate servers and locate individual items inside a server (e.g., files in a file server, pipes in a pipe server, etc.). This name space is largely independent of syntactic issues for path names, such as the interpretation of symbolic links, UNIX "..", etc. It supports user-centered naming through the use of *prefix tables*[12].

I/O: the I/O interface provides a neutral model for the identification and transfer of data. It supports both byte-level and record-level data access, as well as sequential and random access operation. Most issues of synchronization are left to the clients (emulation libraries) for maximum flexibility: when necessary, asynchronous operations are simply implemented with multiple client threads. In addition, various caching and buffer management policies can be implemented within the same I/O framework.

network control: the network control operations deal with the creation and management of transport endpoints, to the exclusion of actual I/O on those endpoints. This interface is largely inspired by XTI[6], with some changes to facilitate sharing of endpoints between multiple clients, and for integration in the uniform name space.

asynchronous notifications: this subsystem can be used by servers to deliver all kinds of notifications to clients, such as UNIX signals, VMS events, etc.

Each of these basic interfaces does not directly define the complete functionality exported by any individual item; rather, they correspond to lower-level services that must be combined to define the complete item. For example, network endpoints typically export operations from both the I/O and network control interface categories.

Many primitives of various target operating environments can simply be implemented with an equivalent primitive from a standard system interface, or with a combination of such primitives. However, it does not seem either possible or practical to define all standard interfaces to incorporate every feature of every conceivable target system. Therefore, there must be exceptions that must be handled with more specialized system primitives. The object-oriented framework provides a sound basis for the definition of such specialized

functions, either through the addition of new specialized interfaces alongside the standard ones for various specialized items, or through the "derivation" of specialized interfaces from the standard ones by adding more complex or modified operations. Since the definitions of all the items are cleanly separated from each other, such extensions can be made with minimal impact on the system as whole.

It is clear that a trade-off exists between defining standard interfaces that represent the union of many different target systems, and resorting to specialization of interfaces to handle some less-common or less-generalizable features. In general, a complete system contains a mix of generic and specialized interfaces, as well as a mix of generic and specialized servers. One significant example of this trade-off is the definition of a collection of attributes for various items, that can be used to implement the UNIX `stat()` operation: the set of all possible attributes and combinations is simply too large. Accordingly, we have defined a "reasonable" set of standard attributes, and rely on additional specialized operations to manipulate all other attributes. Some other examples of OS features that seem best handled through specialization are the combination of multiple I/O channels into one (message queues in UNIX System V, reading out-of-band data inline on BSD sockets), complex protections that do not fit the simple access control list scheme (AFS, UNIX "sticky bit" on directories), special blocking options for `open()` on UNIX TTY's, etc.

Regardless of the resolution of the trade-off between generalization and specialization, it remains the case that the generic interfaces are typically more complex than the corresponding ones in any particular target OS, since they must often support a richer set of features, and tend to have many service options. As mentioned earlier, this complexity is not visible to application programs and is thus not a primary problem, but it does significantly complicate the task of providing correct and complete server implementations of each interface. We have found that a good "standard library" of reusable components or building blocks is crucial to solving this problem.

In addition, in order to maximize flexibility, complex operations are sometimes decomposed into several simpler primitives in the generic interfaces. For example, most UNIX file system operations are implemented with a combination of one or more steps to navigate through the system name space to locate a particular entity in the file system, followed by one or more additional steps directed at the item returned in the previous phase. Similarly, several steps are typically needed to completely set-up a socket (creation, establishment of service options, installation in the user-visible name space). This approach gives rise to two new considerations:

First, whenever the state of the system or item is visible between individual steps of such a sequence of operations, there is a question as to the management and protection of that state. The design of the item interfaces is such that, in most cases, this does not cause significant difficulties. The relevant state information is either naturally made visible and accessible to all (e.g., item references or locks) or it is abstracted away through the use of options and information maintained by the client and provided explicitly with each primitive (e.g., I/O modes and current offset). In one case, however, we have had to introduce a special mechanism: it is possible to create a "reserved entry" in a directory to lock a name until a complete entry can be established. This entry prevents another entry from being created with the same name, but it cannot be looked-up in the traditional fashion. In general, we have considered introducing a general-purpose locking facility to protect groups of operations, but have not found such a facility to be required so far.

Second, and probably more importantly, the use of sequences of operations to perform common tasks raises legitimate performance concerns. This question introduces another design trade-off, to define a set of special "composite operations" to optimize selected functions without unduly compromising the flexibility and reusability of the various servers. The number of these operations must be kept small, because they may in principle have to be implemented by every server participating in every emulation system. The prime example of this question is again provided by the UNIX *stat()* operation: the design defines a composite operation that combines a simple name resolving step to locate a file and a simple operation to retrieve a set of common attributes. This composite operation only covers the most common usage pattern in the system; if the resolving process has to be more complex (e.g. follow mount points or symbolic links), or if special attributes must be retrieved, the clients must fall back on the more primitive operations.

To try to provide another approach to this problem, we are also planning to investigate extensions in the

RPC system to allow "batching" or grouping of several primitive operations in a single high-level invocation without requiring changes to the simple interfaces.

3.3. Modular Services

In order to maximize the overall flexibility, and to take advantage of the potential offered by the idea of reusable services, the design aims to define as many independent servers as possible. Those servers can then be used as a collection of standardized building blocks that can be assembled in various ways to create different systems. In addition to its great flexibility, such an architecture also increases the security and robustness of the system by isolating faulty or potentially malicious components into separate, protected address spaces. Moreover, it sometimes simplifies the implementation of some servers, by allowing the use of multiple instances of one server instead of a single more complicated server (for example, one simple file server for each disk partition). However, such a desire for maximum modularity must be balanced by a number of practical considerations:

- Interactions between servers are more expensive than interactions between modules inside a particular server. The separation of services and the corresponding interfaces must be carefully defined to minimize those interactions. We have tried to limit each basic user-level operation to require the intervention of only one server. There are two main examples where this principle creates difficulties in our prototype: the interaction between the process manager and the TTY server to handle job control, and the general problem of UNIX `fork()`, where the parent emulated process, which is normally in contact with many servers, must arrange for the child process to inherit access to all those servers, along with a consistent client state. We have no solution to the job control problem. We handle the `fork()` problem by defining the system interfaces such that a single association between a client and a server (i.e. a Mach port) can be transparently shared between all the processes in a UNIX process tree, without requiring the explicit intervention of each server. Note that with this scheme, from the perspective of a server, all the client processes that share the same authentication credentials are normally indistinguishable.

  In general, the separation between servers, and between servers and clients, increases the importance of distributed shared state in the system. Much of this difficulty is solved by appropriate selection of the location of each item of information throughout the system (see "Smart Emulation Libraries" below). In addition, we are investigating the possibility of using a general-purpose facility (blackboard service) that allows more-or-less arbitrary data elements to be shared efficiently between any number of servers and clients, according to a schema that can be adapted to the needs of various specific emulations.

- The authentication logic in the system can become complicated, since each server is normally responsible for verifying the identity of its clients independently. This policy also introduces additional difficulties if different authentication schemes are to be used with different servers and if the system must handle mutual suspicion between clients and arbitrary servers. The standard access control and naming interfaces offer basic support for these requirements by providing a good separation between the functions related to normal client access and the authentication function itself: clients may be arbitrarily requested to re-authenticate themselves whenever they start referring to any specific new item, and the particular authentication mechanism to be used may be different in each case.

- Although each server typically performs a different high-level function, there are many low-level functions that must be performed similarly by all servers, such as access mediation, name space management, buffer management, etc. Modules implementing most of these functions are stored in a common library used by all servers, but the overall run-time memory usage of the system is considerably greater than that of an equivalent monolithic system. This problem could be partially solved by the use of a shared library to eliminate duplication of code segments, but it remains the case that much data space is essentially wasted.

In general, the design of the system interfaces is such that clients (emulation libraries) are unaware of which server is handling which item, so that servers can be combined or separated freely without requiring

changes in the clients. The few elements of state that are logically associated with each client-server pair (mostly authentication information) appear to the clients as if they were attached to each individual item, and are transparently replicated and inherited among multiple items managed by the same server as appropriate. However, there is also a need for a global knowledge of the system, to handle operations such as orderly shutdown, startup, administration and maintenance, etc., as well as to keep track of information needed by all components, such as the current time zone, node identity, etc. We are currently building a *system configuration server* to keep track of all the servers in the system and to perform those duties. Finally, the question of implementing global system usage accounting and of enforcing global resource limits has received little attention so far; the main difficulties in this area are the need for servers to identify individual client processes (and not just groups of processes with the same identity), and of efficiently collecting them among a collection of independent servers.

In practice, our BSD prototype contains or will contain the following servers:

Servers that provide services directly visible to application programs (through emulation libraries):

- one or more file servers, handling different file systems (UFS, NFS, AFS, etc.) and/or physical devices.
- a terminal server managing all serial lines and implementing the equivalent of all UNIX tty's and pty's.
- a local IPC server or "pipenet server", responsible for all intra-node emulation-level (non-Mach) IPC: pipes, UNIX-domain sockets, and possibly System V queues and VMS mailboxes.
- a process management server or "task master", to keep track of all emulated process and process groups, and handle signal dispatching.
- one or more network servers implementing socket-like entities for a variety of network protocol families. This server is currently derived from the x-kernel from the University of Arizona[7].
- a device server handling raw access to the hardware devices, grouped in a "/dev" directory.

Servers that support the operation of the other servers:

- one or more *root name servers* tying all the other servers into a single hierarchical name space through a collection of *mount points*.
- a simple authentication server providing trusted translations between client *tokens* and explicit *credentials*.
- a blackboard server managing distributed state stored in pages of shared memory mapped in various servers and client emulation libraries.
- a configuration/admin/startup server that starts all the other servers and provides centralized access to global information.
- various ancillary servers not directly related to the emulation system, for diagnostics, network shared memory, network IPC, etc.

3.4. Client-side Processing and Smart Emulation Libraries

Independent of providing a convenient mechanism for introducing a separation between a generic and a target-specific layer in the system, the emulation library also provides an opportunity to optimize or simplify many client-server interactions by displacing some of the processing required to implement various functions from the system servers into the clients of these services themselves, and to concentrate system state in these clients. This approach is illustrated by three main strategies:

- The design of many service interfaces is such that the client itself is primarily responsible for managing important pieces of information (UNIX file descriptor table, signal mask and handlers, current working directory, etc.) and performing complex functions (pathname resolution, `exec()` logic, etc.).
- Whenever a server grants access for a given item to a given client, a special code fragment or *proxy object*[10] is installed by the run-time system in the address space of that client to act as a local representative for the associated item. This proxy, which is defined and supplied by the server, provides a convenient level of indirection between clients and servers. Simple proxies just forward all client requests directly to the server via Mach IPC, but more complex ones can implement optimizations such as caching item information, or managing a window of shared memory containing file data or other I/O or control information. Proxies are currently statically linked with each emulation library, but we expect that they will eventually be dynamically loaded.
- The emulation library can itself operate as an active element, to simplify and relieve some of the burden on servers. For example, the UNIX emulation library currently contains an active thread to handle incoming asynchronous notifications and transform them into the appropriate UNIX signals. We also plan to use active emulation library threads to implement most forms of asynchronous I/O.

It is clear that there are limitations to the use of client-side processing. There are many system functions that require synchronization and sharing of information or resources between several clients. Although there are mechanisms to handle a number of these problems with minimal server overhead, the need for external agents or servers cannot be completely eliminated. More importantly, the decision to place more responsibility for various system functions in an emulation library that is not protected from incorrect or malicious user programs has obvious implications in the areas of robustness and security. Clearly, the more code and information that is stored in the emulation library, the greater the risk is of accidental or intentional modification, which can lead to very complex failure modes. One interesting example of this problem is presented by the `copyin` and `copyout` routines, that are used in traditional UNIX systems to transfer system call arguments between user space and the protected system space. In our emulated system, these routines may verify the validity of the user buffers when they are invoked and even copy the associated data to separate emulation-space buffers, but there is no way to effectively protect this data from modifications at any time while it is being used inside the emulation library.

In accordance with the goal of supporting very secure system implementations, our design policy is to avoid any optimization that can allow a malicious emulated client to gain unauthorized access to protected resources, or to affect the integrity of another emulated client in ways that would not be possible simply through the use of the "published" application programming interface being emulated. On the other hand, we believe that reasonable compromises are acceptable in trading-off performance against the robustness of individual clients. An incorrect user program may be allowed to corrupt the data structures of its own emulation library. As a result, it may even modify the external behavior of the whole emulated process, but only in ways that could also be achieved with a different (correct) user program. The key element that permits the implementation of such a policy resides in the use of port capabilities for all individual items manipulated by the client: these can be destroyed or even swapped, but never actually forged. However, this approach requires very careful design, and we know of no formal method to guarantee its correct application.

We have also considered a scheme in which a substantial portion of the emulation library is kept separate from the emulated process itself in a different protected Mach task, so that each emulated process is represented by two Mach tasks. However, the communication bandwidth between an emulated process and such a separate library is per necessity lower than that of the fully-shared case, since at least some data must cross address space boundaries, and since control must be transferred between threads. In addition, the cost of creating a new emulated process is typically significantly higher, since two Mach tasks must be created instead of one. Consequently, the decision to use a protected or unprotected emulation library must be made by balancing the added performance of one against the added robustness of the other. We feel that in a system such as UNIX, where `fork()` operations are frequent, the unprotected approach is probably preferable. On the other hand, in a system such as VMS, where process creation is rare, the trade-off may lean the other way. Note that this particular trade-off does not detract from the potential for reuse of server components and generic services

for the emulation of different target systems, since emulation libraries are necessarily specific to each target operating system.

Two other schemes have also been proposed with respect to the location and protection of the emulation library: placing it in kernel space within each emulated task, and displacing all of its function into a single centralized server independent of the emulated processes themselves. Both of these schemes are obviously feasible, but they contradict our basic design decisions to avoid including specialized emulation code in kernel space and to avoid using a centralized coordinating agent, respectively. Consequently, they have not been considered further within the scope of this particular investigation

Independent of these high-level considerations, the increased complexity of smart emulation libraries also creates very significant difficulties at the technical level: increased size, expensive management of client state and complexity due to multi-threading.

The size of a smart emulation library tends to be considerably larger than that of a simple library that blindly forwards all system calls to one or several servers. This results in an increased load on the virtual memory system, both in the form of normal paging activity and in the form of copy faults when creating new emulated processes (UNIX `fork()`).

Smart emulation libraries also tend to contain significant amounts of state that must be inherited or re-created during process creation. Simple memory can be naturally inherited through the services of the Mach kernel, but other elements of state cannot, such as active threads and port capabilities. This introduces potentially significant overheads. In particular, it is clear that a trade-off exists with respect to inheritance of information simply cached in the emulation library, versus simply re-creating this cached information when and if needed. This trade-off has not yet been satisfactorily explored.

Finally, smart emulation libraries naturally need to be multi-threaded, both to support multiple user threads issuing system calls, and to support active threads internal to the emulation library. We have found that the existing implementations of the Mach cthreads library are poorly adapted to operate inside an emulation library. We have had to introduce modifications to handle interactions with threads in the emulated process itself, for correct transfer of thread state at `fork()` time (clean-up stacks and cthread data structures), and to implement an interrupt handling facility for signal delivery. Even with these extensions, multi-threaded emulation libraries are still difficult to write. Furthermore, it does not appear that newer implementations of user-level threads libraries will reduce the magnitude of this problem. In particular, the current trend towards mixing coroutines and kernel-level threads clearly complicates matters, since the emulation system must often still deal with "raw" kernel threads for Mach exceptions and for interrupts.

3.5. Object-oriented Service Library

In order to avoid unnecessary duplication of code, the system design includes a common library implementing many functions that must be provided in several different servers, or in several specialized versions of the same service. Such a library, through its modularity, greatly contributes to the extensibility and ease of modification of the system and also reduces the overall complexity of the entire system implementation. Our prototype contains a growing collection of reusable code fragments or objects; the main areas currently covered are:

- access mediation for arbitrary items (access control lists, user credentials, authentication, etc.).
- simple building blocks to construct a name space collecting all the items managed by one server, and to handle item creation and deallocation when appropriate.
- mapped-file access, including pagers and appropriate proxy objects.
- sequential streams of data following the standard I/O interface (connections, buffering, data transfer across address spaces, etc.).

- simple pathname resolution from the client side, including a user-controlled prefix table and caching of mount points and symbolic links.
- a layer of code to export the standard I/O and naming interfaces from a server based internally on *vnodes*[5].

Considerable effort has been put in the design and implementation of this common library, and we have been served very well by this approach in practice. Some servers (root name server, pipenet server) are constructed almost exclusively from modules in this library. The only non-common code in these servers is a small *main* procedure for startup and a few lines of code to specialize the operation of various modules in the library and supply some operating parameters. This specialization is normally realized through derivation in the class hierarchy defined by the library. Other servers (UFS, NFS, TTY, network) contain a large body of "internal" specialized code, often inherited from other sources. They have been integrated in the system in a matter of hours simply by linking them with an "upper-layer" of code from the common library.

Several of these modules were first developed in earlier versions of our prototype using a straight "C" coding style, but it became quickly evident that this style was poorly adapted to writing the kind of polymorphic, reusable code desired for such a library. The library has since been converted to use object-oriented technology (first using MachObjects[4], then C++), augmented by a powerful remote procedure call package closely integrated with the object-oriented language itself (see below). We have found that this switch in implementation style has considerably simplified the task of writing new modules and integrating them with the rest of the system. Several observations can be made with respect to the use of object-oriented technology in this context[3]

First, multiple-inheritance significantly simplifies the design of the library. We use it in several instances for the combination of independent interfaces for the definition of items, as suggested above. We also use it routinely for the combination of independent pieces of functionality for the implementation of items, for example to add a protection module or a buffer management module to an item representing a file or socket. In addition, in the C++ implementation of the library, we use multiple inheritance to allow classes implementing various functions in the system to inherit both from base classes in a normal "implementation" hierarchy, and from classes in an abstract hierarchy representing the standard system interfaces (used for defining the same interfaces on the client- and server-side).

Second, we need some form of runtime type checking in the language. Although static type checking clearly offers important benefits in terms of program reliability and maintenance, it cannot be used in all situations, specifically for the implementation of method invocations across a client-server boundary. When first resolving a name to obtain a reference to an item in our standard naming subsystem, as with the UNIX `open()` primitive, only the name of that item is known. The actual type and the exact collection of operations exported by that item can only be determined at run-time. This information must then be explicitly obtained by the client upon acquiring or using a proxy for that item. The total number of defined interfaces makes it very impractical to define a single exported type that exports the union of all possible operations. Further, if we allow for maximum flexibility and the definition and integration of new servers and interfaces over the life the system, even the complete set of possible operations may not be known in advance. With the MachObjects system, this problem is moot since all method invocations are checked at run-time. In the C++ system, we use static type checking wherever possible, coupled with a package similar to the NIH class library[2] to provide the required extended functionality for run-time type checking.

Lastly, it is very useful to be able to manipulate classes and methods as first-class abstractions in the language. The proxy mechanism, which is based on the run-time specification of a class to instantiate in a client, obviously creates a need to refer to classes explicitly and directly. In addition, the RPC package integrated with the object-oriented programming environment manipulates method references directly at run-time to transparently forward some invocations without the need to generate explicit stubs, thereby greatly enhancing the flexibility of this system and speeding-up the design and prototyping process. Method references are also manipulated by the access mediation subsystem, which uses a simple table lookup to determine if a given method invocation is to be allowed for a given caller and object combination. As is the

case for run-time type checking, these features are handled trivially in MachObjects, and with the help of an extension package in C++.

3.6. Hiding Complexity in the Tools

In addition to the common library mentioned above, the design also attempts to reduce the complexity directly visible to implementors by aggressively hiding functionality in a collection of powerful run-time and library tools. These tools can be used largely as opaque "black boxes"; they create another degree of fine-grain layering in the overall system design. The design of many such complicated tools could easily constitute a research project in itself, considerably beyond the scope of our current goals. Accordingly, our focus is not on developing ideal general-purpose solutions in this area, but on solving the specific problems and issues raised in our design, and only as we encounter them. Our success with this approach so far encourages us to continue in this direction.

A first example of this approach is the use of the extended cthreads library described above, that takes care of many of the difficulties inherent in the implementation of emulation libraries. The main complexity introduced at this level is a facility used to interrupt pending operations in the emulation library. This facility must be invoked whenever the reception of a UNIX signal must cause the execution of a system call to be prematurely aborted. It includes a relatively general exception handling mechanism, and it interacts with the RPC system to propagate such interruptions to various servers as appropriate. Another example is the set of extensions to the object-oriented run-time system described in the previous section.

More importantly, the last and probably most far-reaching example is a sophisticated remote procedure call package integrated with the object-oriented language used for the implementation of the prototype. This RPC package is the key to many of the features mentioned in the rest of this overview. It transparently converts local method invocations into RPC's as appropriate, and makes it very easy to define new RPC's. It handles interruptions of arbitrary RPC's through a special message protocol that is implemented in all servers. It automatically takes care of transferring item references across address spaces and of instantiating the correct proxy objects. Finally, it completely hides all details of the management of ports and Mach IPC semantics from its users, including taking care of automatic garbage-collection of item references using the "no-more-senders" facility of the Mach IPC system.

Not surprisingly, this RPC package turns out to be very complex, and to be a significant factor for the performance of the prototype. However, we feel that most of this complexity is essentially unavoidable, and would simply have to be distributed through other parts of the system if it was not concentrated in the RPC module. Therefore, we plan to continue to extend the RPC package in directions that we think will improve the performance and flexibility of the overall system. Some proposed experiments include automatic initialization of complex proxies, the use of polymorphic abstract data types as arguments for certain calls, and batching of operations to reduce the total number of messages used in the system.

4. Status and Evaluation

The main observation to be made at this stage is that the overall architecture appears to be sound and effective. The current system prototype for BSD does work with completely modular services and has already reached a relatively high level of functionality and practical utility. The flexible technology for interface definition is adequate and we expect that it will continue to be successful as we refine that prototype. The framework for integrating servers into the system, centered on the naming and access mediation components, has proven to be extremely useful. Finally, the use of modular servers has been and continues to be invaluable in helping with the incremental construction and debugging of the system: we routinely restart or replace individual servers while the whole system is operating, and debug them "on-the-spot" using the Mach task and thread control facilities.

The BSD prototype currently implements all the essential functionality for `fork()` and `exec()`, general file access services, signal and process management (including interrupted system calls), TTY management, pipes, and UNIX-domain and IP/UDP sockets. In practice, the system can be started or "booted" on top of the *POE* low-level emulator, run an emulated login process, start a shell and run various editors and common UNIX commands, up to and including compilation of programs. Still missing are TCP sockets, raw device access, and a number of more specialized or less frequently used features in the above subsystems, for example asynchronous I/O, set-uid `exec()`, sharing of file descriptors between processes, resource control, etc. We continue to add new features in these areas and to integrate new facilities. The major implementation activity at the present time is a conversion to use C++ as the standard implementation language.

We have not made any significant efforts to improve the robustness and performance of the prototype until its level of functionality was sufficient to make such efforts meaningful. We feel that this point has now been reached, and detailed evaluation and tuning should begin in earnest as soon as the conversion to C++ is complete. A first short-term objective is to make the prototype self-hosting.

The degree to which various elements of this system are reusable is still a matter of discussion, but early observations are encouraging. We feel that the design of many servers and high-level components, coupled with the generic service interfaces, has very good potential for flexibility and reusability under various operating conditions. In addition, the smaller-level reusability of the pieces of the common service library has already been largely demonstrated; we believe that the current prototype could not have been built without it. A desirable further step in this direction should be the construction of a similar library for the implementation of emulation libraries.

On the negative side, the size and complexity of the current prototype are clearly causes for concern. The memory usage of the multiple servers is large, and the load that they impose on the machine is important. Re-compilation of the prototype is slow, due to the volume of code involved. Several of our low-level mechanisms and sub-systems are very large and seem overly complicated, for example the facility to handle interrupted system calls and all the logic around UNIX `fork()`. Hopefully, the magnitude of these problems will decrease as the design and its implementation become more mature, but it is not clear if they will ever be completely resolved.

Finally, we cannot yet draw conclusions on the experiment with the use of "smart" emulation libraries in general or the degree to which this general approach can and should be applied. There are practical examples of other systems where this approach has been useful[1], but none that goes as far as the present design. The difficulties in terms of complexity, robustness and increased *fork()* overhead cannot be ignored. We will reserve final judgment on this experiment until more experience has been gained with implementing and using systems that rely on this aspect of the architecture.

5. Conclusion

We are conducting experiments and proposing a number of techniques with the goals of helping the development of emulation systems, and of minimizing the duplication of effort for the implementation of several independent emulations. There is no question that achieving these goals is highly desirable; the present effort, although still incomplete, shows that there are many opportunities to make significant progress in this direction. Furthermore, several of the proposed general-purpose components or mechanisms constitute significant practical steps toward improving the current state of the technology, and the results from our BSD prototype encourage us to continue this effort. We expect that the experiences from this investigation, combined with the lessons from other efforts in the same area, will eventually lead to the development of a complete practical framework for the implementation of many emulators, and maybe to the emergence of a new software industry for the production of replaceable, modular components for Mach-based systems.

Acknowledgments

Many people at CMU and at the Research Institute of OSF have participated in various stages of the design of the system and the implementation of successive versions of the BSD prototype. Beside the authors, they are: Robert Baron, Alessandro Forin, Jeffrey Heller, Michael Jones, Keith Loepere, Douglas Orr, Richard Rashid, Franklin Reynolds and Richard Sanzi. In addition, the whole Trusted-Mach team at Trusted Information Systems has often contributed many valuable insights.

References

[1] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. Unix as an application program. In *Proceedings of the 1990 Summer Usenix*. Usenix, June 1990.

[2] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley and Sons, 1990.

[3] Paulo Guedes and Daniel P. Julin. Object-oriented interfaces in the Mach 3.0 multi-server system. To appear in the proceedings of the Second International Workshop on Object-Orientation in Operating Systems, Palo Alto, 1991.

[4] Daniel P. Julin and Richard F. Rashid. MachObjects. Internal document, Mach Project, Carnegie Mellon University, 1989.

[5] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the 1986 summer Usenix*, pages 238–247. Usenix, 1986.

[6] Open Software Foundation. *OSF/1 Network Programmer's Guide*, 1990.

[7] Larry L. Peterson, Norman C. Hutchinson, Sean W. O'Malley, and Herman C. Rao. The *x*-Kernel: A platform for accessing internet resources. *IEEE Computer*, 23(5):23–33, May 1990.

[8] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Daniel Julin, Douglas Orr, and Richard Sanzi. Mach: A foundation for open systems. In *Proceedings of the Second Workshop on Workstation Operating Systems*, pages 109–113. IEEE Computer Society, September 1989.

[9] Richard Rashid, Gerald Malan, David Golub, and Robert Baron. DOS as a Mach 3.0 application. To appear in the proceedings of the Second USENIX Mach Symposium, November 1991.

[10] Marc Shapiro. Structure and encapsulation in distributed computing systems: the Proxy principle. In *The 6th International Conference on Distributed Computing Systems*, Boston (USA), May 1986.

[11] Trusted Information Systems, Inc. Trusted Mach system architecture. Internal Report, April 1990.

[12] B. Welch and J. Ousterhout. Prefix tables: A simple mechanism for locating files in a distributed system. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 184–189. IEEE, May 1986.

DOS as a Mach 3.0 Application

Gerald Malan, Richard Rashid, David Golub, and Robert Baron
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue Pittsburgh, Pennsylvania 15213

(412) 268-8744

Internet: grm@cs.cmu.edu, rfr@cs.cmu.edu, dbg@cs.cmu.edu, rvb@cs.cmu.edu

Abstract

We have implemented support for the DOS operating system on top of the Mach 3.0 kernel. This support included machine-dependent kernel modifications for the i386/i486 architecture to handle virtual 8086 mode, a multithreaded emulation of the IBM PC's VGA display and I/O devices, direct support for a number of common DOS functions and frequently loaded DOS drivers, and code to integrate DOS functionality with the existing 4.3BSD Unix Server. The resulting system allows multiple virtual DOS environments, supports DOS versions 3.1 to 5.0, and is capable of running common DOS software including performance sensitive PC entertainment software such as Wing Commander – a high-speed space combat simulation system.

Many lessons were learned during the course of this work. DOS stresses a number of Mach features infrequently used by Unix emulation. The timing and latency demands of DOS applications, especially those with animation and sound, are dramatically different from those of typical Unix applications. Because most DOS programs interact intimately with the underlying PC hardware, DOS emulation expanded our knowledge about the behavior of Mach 3.0 when asked to provide a virtual machine environment rather than pure client/server support. In particular, quirks of the PC architecture – such as support for the so-called "high memory area" above 0x100000 – had to be precisely emulated.

This paper describes our implementation, its capabilities and limitations, and the lessons learned about Mach in the course of our development effort.

1. Introduction

Approximately one year ago we undertook, as an experiment, the implementation of support for the i386/i486 family's "virtual 8086" mode and the DOS operating system. We were encouraged to begin this work by the success already achieved in the implementation on the Macintosh of support for the MacOS and Multifinder under Mach 2.5. We felt that DOS and the implementation of a virtual x86 machine would present a new set of challenges for Mach's abstractions and implementation.

Today Mach 3.0 DOS support consists of:

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDS972-90-C-0035 and in part by the Open Software Foundation (OSF). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, OSF, or the U.S. government.

- machine-dependent kernel support for virtual 8086 mode
- a multithreaded emulator task which provides:
  - virtual Industry Standard Architecture (ISA) I/O devices;
  - BIOS emulation;
  - DOS ISA device drivers;
  - emulation of common DOS extensions such as Expanded Memory (LIM 3.2), Extended Memory, High Memory and Upper Memory Block management (XMS 2.0);
  - direct implementation of many common DOS functions; and
  - software to integrate DOS with Mach's existing Unix server.

The system supports DOS versions 3.1 to 5.0 and runs Windows 3.0 in real-mode. We have successfully run over 100 DOS applications ranging from business software such as Lotus 123 and Microsoft Word to entertainment software such as Wing Commander, Space Quest IV, King's Quest V, Populous, and the Microsoft Flight Simulator. Many of these applications are extremely demanding in terms of both graphics performance for VGA displays and sound generated by commercial monaural and stereo sound boards.

This paper describes our implementation, our experiences, the lessons we have learned and the potential impact of our work on future Mach development.

2. Overview of DOS

A limited knowledge of DOS and the Intel 80386 processor is needed to understand the Mach DOS support. There are three main areas of interest: the way DOS programs call the operating system, the organization of a DOS machine's physical memory, and the operation of the Intel 80386 processor's Virtual 8086 mode.

Two operating system modules are used by DOS programs for system support, the BIOS ROM and the DOS kernel. The BIOS is a set of ROM routines that support device specific system calls. The DOS kernel is located in RAM and supports a general operating system interface. The DOS kernel uses BIOS routines to perform many operations, but DOS programs frequently bypass the DOS kernel and directly call the BIOS.

DOS programs perform system calls by loading call parameters into machine registers and then generating a numbered exception. The registers used to pass parameters vary depending on the system call. Exceptions are generated by DOS programs with the `INT` machine instruction. The `INT` instruction takes the exception number as an argument which is used as an index into the interrupt vector table located in low memory. The interrupt vector selected by the `INT` instruction determines which interrupt service routine will handle the exception. Different exception numbers are used by different DOS and BIOS system calls. DOS provides a general operating system interface that only has one of these exception "entry points." The BIOS routines are grouped by device services, so there are many different BIOS exception numbers (eg. the BIOS video services are all entered via exception number 0x10, the disk routines all use exception 0x13).

The address space of a typical DOS machine is shown in figure 1. Key areas are the interrupt vector table, the BIOS scratch pad RAM, and the memory above 0xA0000. The interrupt vector table contains the addresses for the exception handling routines, referred to below as Interrupt Service Routines or ISRs. DOS programs often manipulate these ISR addresses to redirect different interrupts into their own ISRs. The BIOS scratch pad RAM is used to keep device state information

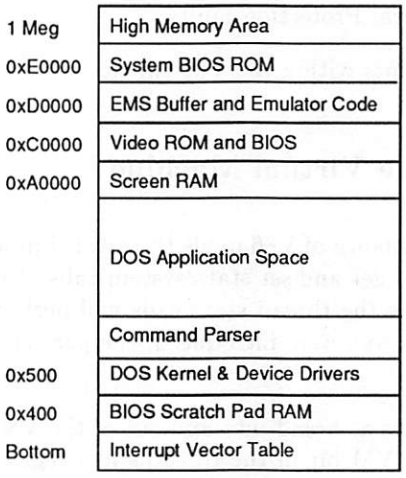


Figure 1: DOS Application Memory Organization.

for several important devices. It is in this scratch pad RAM that the BIOS keyboard routines keep the queue of incoming keyboard characters, the video driver keeps the scan line state, and the disk driver saves motor status and small input buffers. The area above 0xA0000 contains the video RAM, Extended Memory Manager swap buffers, BIOS ROM routines and High Memory area.

Central to the Mach DOS support is the Virtual 8086 (V86) operating mode of the Intel 80386 processor. When the processor is in this V86 mode, it interprets memory and executes instructions as if the machine were a native DOS 8086 machine. A set of *privileged* instructions encountered by this V86 machine generates faults that are passed to a protected mode monitor. In the Mach DOS system, a thread operating in the V86 mode generates General Protection (GP) faults that are passed to the Mach Kernel when a privileged instruction is executed. The instructions that are privileged under V86 mode are those that: manipulate the interrupt state of the machine, generate or return from interrupts, and access privileged I/O ports.

3. The Organization of DOS as a Mach Application

The implementation of DOS as a Mach application consists of two pieces, support within the Mach kernel, and a Mach task used as a DOS emulator. The Mach kernel manages threads running in V86 mode and the faults that they generate. The emulator task provides DOS and BIOS system call support. It is important to note, that the DOS Emulator runs the native DOS kernel in a V86 mode thread and does not attempt to reimplement all of the DOS and BIOS system calls.

3.1. Microkernel support for DOS

The Mach kernel provides management for V86 mode threads, and the interface between a V86 thread and its associated Emulator task. Specifically the kernel provides:

- V86 thread creation and maintenance through existing Mach system calls
- V86 thread interrupt flag management using a "virtual" interrupt flag

- Handling of simple V86 General Protection faults
- Simulation of external interrupts within the V86 thread.

3.1.1. Manipulation of the Virtual Machine

Support for the creation and maintenance of V86 mode threads is implemented in the Mach kernel in the form of extensions to the thread get and set state system calls. Different types of thread *flavors* determine which kinds of operations the thread state calls will perform. The existing *mode* thread flavor was modified and two new flavors were introduced, the *port map* and *V86 assist* flavors.

- The *mode* flavor is used to take a thread into and out of the V86 operating mode. To turn on the V86 operating mode, the VM bit in the thread's flag register is set. To turn off the V86 mode, the VM bit is cleared.
- The *port map* flavor enables or disables the threads access to a given ISA I/O port. In this way, direct access to the VGA registers can be given to the V86 thread to speed up video intensive DOS programs.
- The *v86 assist* flavor accesses the kernel's V86 support data structures that contain a pointer to the Emulator task's interrupt queue, and the value for the V86 thread's "virtual" interrupt flag (see below).

3.1.2. Managing the Interrupt Flag

Many of the V86 thread's privileged instructions attempt to manipulate the machine's interrupt status, requiring the kernel to implement a "virtual" interrupt flag for each V86 thread. The kernel alone is allowed to enable or disable these interrupts, yet DOS programs all assume that they are in total control of the machine. Therefore it was necessary to modify the Mach kernel to provide a "virtual" interrupt state to the V86 thread. Since the V86 thread must perform a privileged instruction to inspect or modify the interrupt flag, the Mach kernel can always replace the true interrupt bit with this "virtual" interrupt flag. Space was made in the V86 thread's process control block to hold the flag, and the kernel was modified so that the `thread_get_state` and `thread_set_state` routines would access the "virtual" interrupt flag instead of the true one.

Additional kernel support is also required because of the peculiar way in which 80x86 processors delay the setting of the interrupt bit in the flag register after an STI instruction. The STI instruction doesn't set the interrupt flag during its execution, but delays the setting of that flag until after the *next* instruction finishes executing. Some DOS programs rely on this feature, so its emulation was critical. To solve this problem, the trace flag is set in the V86 thread's flag register during the STI instruction's emulation. Possibly by design, the trace exception then occurs at the same point that the interrupt flag bit should be set. When this trace exception occurs, the kernel sets the interrupt bit in the "virtual" flag register and clears the trace bit in the V86 thread's flags.

3.1.3. General Protection Fault Handling

The privileged instructions that are generated by the V86 thread and are not DOS or BIOS system calls are emulated in the Mach Kernel. These instructions are the set and clear interrupt (STI and CLI), push and pop of the flag register (PUSHF and POPF), and the return from interrupt (IRET) instructions. The instructions are emulated in the kernel as follows:

- CLI: Clears the "virtual" interrupt flag
- STI: Sets the "virtual" interrupt flag as described above
- PUSHF: Pushes the flag register onto the V86 thread's stack. The "virtual" interrupt flag bit is masked onto this value.
- POPF: Pops the flag register from the V86 thread's stack. The "virtual" interrupt flag bit is set to the value from the stack.
- IRET: The kernel pops the flag register from the stack like a POPF, and then pops the V86 thread's instruction pointer from the stack.

3.1.4. Simulating Interrupts Within the Virtual Machine

The Mach kernel is responsible for generating external interrupts within the V86 thread that have been queued by I/O threads in the Emulator task. Using the data structures in the V86 thread's pcb, the kernel determines whether or not there are any external interrupts that need to be delivered to the V86 thread.

To allow for smoother execution, the kernel interrupts the V86 thread sequentially. That is, it waits until the last kernel interrupt has completed before it generates another one. The kernel sets a flag when it generates an interrupt within the V86 thread and clears it when an interrupt return instruction is emulated. While the flag is set, the kernel does not generate any further external interrupts.

3.2. The Emulator Task

The Emulator task is a multithreaded Mach task that supplies input to the V86 thread and provides emulation for DOS and BIOS system calls. Five threads execute simultaneously in the Emulator task: the Monitor, V86, Timer, Mouse, and Keyboard threads. The Monitor thread initializes the DOS support and provides system call emulation. The V86 thread executes the DOS code in virtual 8086 mode. The Timer, Mouse and Keyboard threads all generate the input and interrupts that drive the DOS applications.

The first 1.15 Megabytes of the task's memory is set up by the Emulator to resemble a typical DOS machine. This is accomplished by setting up the interrupt table, scratch pad RAM, screen RAM, BIOS ROM, and high memory area as shown in figure 1. Several "snapshot" files are used to provide the DOS interrupt table and BIOS scratch pad RAM areas. These files are created during a one time setup session. To give fast access to the screen and BIOS ROM routines, the machine's physical memory is mapped onto the task's virtual address space using Mach system calls. Due to the segmented addressing scheme, the 8086 can generate addresses with up to 21 bits, 64 Kbytes above one megabyte. DOS programs expect the 64K High Memory Area above the one Megabyte limit to wrap back around to the beginning of physical memory. The Emulator maps both the High memory area and the first 64K bytes of low memory to the same place in the task's virtual address space.

3.2.1. The Monitor "INT" handler thread

The Monitor thread, formally the V86 thread's exception handler, provides emulation for DOS and BIOS system calls as well as support for privileged IN and OUT instructions. The system call

emulation is broken down into groups depending on the type of service being emulated. Some of these major groups are the BIOS disk, video, mouse, and memory expansion routines as well as the DOS calls related to console I/O.

A secondary function of the monitor thread is to support `IN` and `OUT` instructions that attempt to access privileged I/O ports. To enable an I/O port for access, the emulator must use the `thread_set_state` call using the *port map* flavor described above. The majority of these "privileged" I/O instructions are executed by DOS interrupt service routines. By catching these privileged I/O instructions and supplying artificial return values, the Monitor thread is able to support several types of DOS device drivers including the Windows 3.0 Logitech Mouse driver and many types of DOS keyboard drivers.

- **Emulator Disk Support**

  The Emulator's disk support provides two main functions: emulation for BIOS disk system calls, and access to the unix file system as a Network DOS drive. By emulating the BIOS disk routines, we in effect emulate the DOS disk related system calls which all use BIOS routines to perform the actual disk I/O.

  Access to physical disk devices and to UFS files is provided by the Emulator. Physical devices and UFS files are both used as DOS file systems. The Emulator uses the Unix raw device files to access the hard disk's DOS partitions and DOS file systems on floppy disks. Unix files are also used by the Emulator as dos file system images.

  The Unix file system is also used as a Network DOS disk drive. The Emulator uses the DOS Network Redirector, a group of DOS system calls to achieve this. The Network Redirector is used by DOS to access file systems that do not use DOS file allocation tables. With the Network Redirector interface in place, the V86 thread can access any file within the Unix file system.

- **Emulator Video Support**

  The Emulator keeps track of the VGA video state, emulates BIOS video routines and redirects some video exceptions back into the V86 thread. The Emulator saves the VGA state at startup time and restores it upon exit. This video state consists of VGA register settings and the contents of the VGA's video RAM banks. Video state for each of the VGA's video modes is stored in a file. The Emulator uses the information in this file to change the VGA's state when a mode change is requested by a BIOS system call. This VGA information file is created during the setup session described above. A copy of the machine's default font is also kept in a file that is used with VGA text modes.

  To emulate the BIOS video routines, the emulator directly writes to the VGA registers and video RAM areas. If the emulator does not support a video routine, it passes the routine back into the VGA BIOS for the V86 thread to execute. These redirected routines generally do not require the VGA BIOS to know a previous state and will not crash the video subsystem.

- **Emulator Mouse support**

  The Emulator task supports all the functions of a DOS memory resident mouse device driver. It does this using both the Monitor and Mouse threads. The Emulator's mouse driver complies with version 6.0 of the Microsoft Mouse specification. This specification is comprised of both a set of Mouse related system calls, and hooks that allow DOS programs to supply their own mouse event handlers. The Monitor thread emulates these Mouse BIOS calls while the Mouse thread provides support for the user supplied mouse event handlers.

- **Emulator Expansion support**

  Several DOS expansion specifications are supported by the Emulator including the EMS 3.2 and XMS 2.0 specifications. Both of these specifications, like the Microsoft Mouse specification

mentioned above, are collections of system calls that extend DOS and BIOS functionality. In 1985 Lotus, Intel and Microsoft agreed to support a standard that increased the amount of memory DOS applications could access. This standard, supported by the Emulator task with the Mach virtual memory subsystem, is known as the LIM EMS.

The Expanded Memory Specification (EMS) allows DOS programs to access memory on special hardware boards by switching this memory in and out of certain banks in the DOS memory space. DOS programs access memory in this special hardware by gaining handles to certain expanded memory segments. The Emulator task emulates this behavior by `vm_allocating` memory in the task's address space to use as this special expanded memory and then moving this memory in and out of the special DOS memory banks whenever necessary.

To allow DOS programs access to the conventional memory in the machine above the one megabyte boundary, the eXtended Memory Specification (XMS) was created. This specification also is a collection of BIOS system calls that switch blocks of extended memory with blocks of memory in the addressable DOS area. This is implemented on native DOS machines with special memory drivers that take the machine into and out of protected mode to gain access to the extended memory. The Emulator allocates virtual memory in the same way that it does for the EMS specification and copies this memory into and out of the DOS areas on demand.

3.2.2. The V86 thread

The V86 thread is the only thread that operates in virtual 8086 mode. This thread executes the DOS application and kernel code that generates the general protection faults. These faults are the privileged instructions intercepted by the Mach kernel.

3.2.3. I/O threads

Three I/O threads provide data and external interrupts for DOS programs executing in the V86 thread: the Timer, Mouse and Keyboard threads. These threads provide input for two types of DOS programs: *well-behaved* programs that get their information from BIOS data structures, and *ill-behaved* programs that bypass the BIOS and redirect external device interrupts into their own code. Well-behaved programs access the BIOS data structures through DOS or BIOS system calls. Ill-behaved programs, on the other hand, redirect the notification interrupts away from the BIOS device drivers and into their own ISRs. These Ill-behaved interrupt service routines directly access the input device's I/O ports to gather the input data. To determine which types of programs are executing within the V86 thread, the I/O threads check the interrupt vector table for redirected vectors.

The Monitor and the I/O threads use shared data structures to provide input data for well-behaved DOS programs. The input data for these programs is inserted into the shared data structures by the I/O threads, and is extracted by the monitor thread. In general, a DOS or BIOS system call will direct the monitor thread to place the input data into the V86 thread's registers and then exit to continue the DOS program's execution.

When providing data to an ill-behaved DOS program, an I/O thread queues a notification interrupt, and saves the device input data in an interthread data structure. After the ill-behaved program's ISR has been activated by the interrupt, it will read data from the input device using the privileged `IN` instruction on the device's I/O port. When the Monitor thread catches this attempt to access the privileged I/O port, it will return the data stored previously by the I/O thread as the data from the I/O port, completing the transaction.

The I/O threads use an *interrupt generation table*, shared with the Mach kernel, to generate the external interrupts within the V86 thread. During initialization, the Emulator passes the address of this table to the Mach kernel using the *v86 assist* flavor of the `thread_set_state` system call described above. This shared data structure contains a count and vector number for all the different types of external interrupts that the I/O threads can generate. When an I/O thread wants to generate an interrupt, it increments the interrupt's count field in this table, which causes the Mach kernel to generate the appropriate external interrupt.

Descriptions of the three I/O threads follow. Some of the most difficult problems encountered during the implementation of the DOS Emulator were associated with the Mouse and Keyboard threads. For this reason a greater depth of detail has been provided in these areas of the paper.

- **The Timer Thread**

  Two DOS timer interrupts are generated by the Timer thread: the Time of Day interrupt, and the DOS timer tick interrupt. On a native DOS machine, the Time of Day interrupt is generated by a clock chip connected to interrupt request line zero. The BIOS's Time of Day interrupt service routine generates 18.2 DOS timer tick interrupts per second. The DOS kernel keeps system time using these timer *ticks* in the BIOS scratch pad RAM area.

  Timeouts on Mach ports periodically wake up the timer thread to check the V86 thread's timer interrupt status. When the timer thread wakes from this small timeout value, it sets the system time in the scratch pad ram area and checks the two timer interrupt vector table values. If either of these vectors have been redirected from the values set at the Emulator's initialization, the timer thread queues an appropriate interrupt in the interrupt generation table that triggers an external interrupt by the Mach kernel.

  Many DOS programs require time of day interrupts in shorter time segments than the Mach kernel can provide. These interrupts are needed in small time slices to drive the PC speaker and other sound devices. To accomplish this, the Timer thread calculates and queues the number of timer interrupts that should have occurred during it's timeout period. All of these interrupts are delivered by the kernel in short succession, generating the correct sound output.

- **The Mouse Thread**

  Depending on the type of mouse input expected by the V86 thread, the Mouse thread emulates either a mouse device driver or a mouse connected to a serial line. When emulating a mouse device driver, the Mouse thread stores information in data structures shared with the Monitor thread, queues external interrupts, and updates the mouse pointer. To emulate a serial line mouse, the Mouse thread queues interrupts on the serial line for the V86 thread.

  Acting as the mouse device driver, the Mouse thread's main tasks are to keep track of the mouse's status, to update the mouse pointer in the VGA screen memory, and to provide interrupts for user supplied mouse event handlers. To keep track of the mouse's status, the Mouse thread updates internal variables when the mouse moves or generates button events. When the mouse pointer is enabled by Mouse BIOS system calls, the Mouse thread must display and track the mouse on the screen.

  By far the most interesting aspect of the Mouse thread's device driver support is the implementation of the DOS application-supplied mouse event handlers. The Mouse BIOS specification allows DOS applications to specify a routine that is called by the resident mouse driver when a user defined mouse event occurs. When this mouse event occurs, the mouse device driver loads the machine registers with mouse status information and then activates the application's routine using a `FAR CALL` instruction. Since the Mach kernel is used to asynchronously stop and redirect control flow within the V86 thread, the Mouse thread must use it to perform the switch to the user event handler. The Mouse thread uses an undefined interrupt vector as a special Mouse interrupt vector that calls the DOS event handler. This poses two problems for the Mouse thread:

1. There must be a way to load the V86 registers with the correct calling parameters before control is given to the user event handler.
2. There must be some routine used as an intermediary that will receive the user event handler's **FAR RET** call and will restore the previous program's state.

To solve these problems, the Mouse thread uses two helper routines that are placed in the DOS application's address space during the Emulator's initialization. The first problem is solved by the first helper routine, which is invoked by the special Mouse interrupt generated by the Mach kernel. This first helper routine invokes the Monitor thread to load the calling parameters into the V86 thread's registers and to set up the stack for the DOS event handler's **FAR RET** instruction. The second helper routine, the address of which has been placed on the stack, is invoked when the user event handler finishes. This second helper routine restores the interrupted program's register state from the stack and returns control of the V86 thread back to the place where the Mach kernel's interrupt occurred.

The mouse thread also supports the Windows 3.0 Logitech serial mouse driver by simulating the behavior of the Logitech mouse and the machine's COM port. Windows is the only program that bypasses the emulator's internal mouse driver and demands to use its own. To simulate the behavior of a Logitech serial mouse, the Mouse thread queues interrupts that the Window's device driver services by accessing privileged I/O ports. The Mouse thread keeps track of the serial mouse and its COM port by stepping through a state machine, the transitions of which occur in response to privileged I/O instructions on the COM device's I/O ports. Although it would be possible to support any type of mouse driver by expanding this state machine, no advantages would be gained over the internal mouse driver already present in the Emulator.

- **The Keyboard Thread**

The function of the Keyboard thread is to gather raw keyboard data and to pass that data on to the V86 thread. Depending on the type of DOS program that is executing, the Keyboard thread either puts the keyboard data into a queue or passes the raw scan code to the DOS program's keyboard interrupt service routine. In order to understand the way the emulator handles keyboard data, a knowledge of the BIOS's default keyboard ISR, the scratch pad RAM's keyboard queue and the four general strategies DOS programs use to gather keyboard data is essential.

In a DOS system with *well-behaved* DOS programs, the BIOS keyboard ISR intercepts keyboard interrupts and places decoded characters into the scratch pad RAM's keyboard queue. This *normal* DOS system is one in which the keyboard interrupt vector has not been redirected. Well-behaved DOS programs get their only keyboard input from this queue. They can either use BIOS system calls to indirectly manipulate the queue or bypass the BIOS and access the keyboard queue directly. Some of the DOS programs that perform keyboard input in this manner are the command.com parser, Borland's Turbo C editor, Lotus 123, edlin.com, and debug.com.

Ill-behaved DOS programs perform keyboard input by redirecting keyboard interrupts into their own code. There are three types of programs that gather keyboard input in this way. These are:

- A *Keyboard Hog* program redirects keyboard interrupts into its own code and does not share this information with other DOS programs. This type of program does not place character information into the scratch pad RAM keyboard queue and does not pass the interrupt to another interrupt service routine. In effect it captures the keyboard input all for itself. Epsilon, an Emacs-like text editor, is an example of this type of program.
- A *Terminate and Stay Resident* (TSR) program redirects the keyboard interrupt into its own code like the hog, but after finishing with the scan code, it jumps to the ISR it replaced, in effect passing the interrupt down a chain of DOS interrupt service routines. The DOS program using this strategy would not use the scratch pad RAM's keyboard

queue, but would allow the BIOS keyboard ISR to update the queue through the chaining mechanism.

- The *Middleman* program uses both interrupt chaining and the scratch pad queue for keyboard input. This type of DOS program redirects the keyboard interrupt into its own ISR, chains the interrupt to the old keyboard ISR, and then uses BIOS system calls for character input. This allows DOS TSR programs to filter the character stream before it reaches the program, but allows the program to screen some keyboard events from the TSR programs. Examples of Middleman programs are Microsoft Word 5.0, Microsoft Windows 3.0, Wing Commander, Populous, and the Microsoft M text editor.

The Keyboard thread supports all four types of DOS programs. The well-behaved DOS programs are easily supported by putting characters into the scratch pad's keyboard queue upon arrival. The ill-behaved DOS programs are supported with the aid of a helper routine used as a smart ISR located in the DOS application's address space. When executed, the helper routine puts a character located in its data area into the scratch pad's keyboard queue. At startup time the Emulator saves the address of this helper routine in the keyboard interrupt vector so that programs that chain the keyboard interrupt will call the helper routine.

When the keyboard interrupt vector is redirected, the Keyboard thread queues the scan code in an emulator data structure, then queues a keyboard interrupt in the interrupt generation table. The keyboard interrupts are handled differently than other external interrupts in that they are generated by the Emulator task instead of the Mach kernel. This is done so that the emulator can pass the decoded character to the helper ISR in the DOS application space before the interrupt occurs. This way any programs that chain the keyboard interrupt will eventually place the decoded character into the keyboard queue.

3.2.4. Locking issues

There are many objects shared between the Emulator's threads that can only be manipulated by one thread at a time making the ability to lock these objects important. The objects that can only be accessed sequentially are the V86 thread's state, data structures shared between I/O threads and the Monitor thread, and the interrupt generation table that is shared between the I/O threads and the Mach kernel. To lock these objects, the emulator uses two mutual exclusion objects: a critical section lock and a mouse lock.

To insure that the V86 thread's state is not changed by more than one thread at a time, threads outside the kernel use the critical section lock and the the V86 thread's *resume* flag. The I/O threads take the critical section lock before attempting to suspend the V86 thread so that only one of them is able to modify the V86 thread's state. The emulator's I/O threads must then suspend the V86 thread with the Mach `thread_suspend` system call before they can modify its state. Once the I/O thread has the critical section lock and has suspended the V86 thread, it checks the Resume Flag bit in the V86 thread's flag register. When this bit is set the V86 thread has generated an exception and is being serviced by either the Mach kernel or the emulator's Monitor thread. Since the Mach kernel's V86 support routines are only invoked when the V86 thread is running there is no chance that one of the I/O threads is modifying the V86 thread's state when the Mach kernel's routines are running.

The Emulator's threads must acquire the critical section lock before they can access data structures shared between themselves. The mouse lock is used to guarantee the integrity of the data structures shared by the Monitor thread and the Mouse thread. The critical section lock is also used by the I/O threads to preserve the interrupt generation table's integrity.

4. Integrating DOS with the Unix server

As of this writing, the DOS support relies heavily upon features provided by the 4.3 BSD Unix server in areas including I/O device access, DOS file system implementation, and the management of multiple DOS machines. The DOS emulator task accesses the machine's keyboard, mouse and disk devices through Unix server system calls. Two approaches to accessing DOS file systems are implemented using the Unix server's system calls. Finally, the ability to create and schedule multiple DOS machines also depends on functionality provided by the Unix server. In the future, work will center on replacing the functionality provided by the 4.3 BSD Unix server with native Mach support.

4.1. Unix support for DOS I/O Devices

The Keyboard, mouse and disk devices are all accessed by the emulator task by using Unix system calls. The only I/O device that is not controlled with Unix calls is the display. In general, for input-only devices such as the keyboard and mouse, an I/O thread opens the Unix device file and enters a loop that performs a read select on the device's file descriptor before reading data from the device. The select is performed to check whether or not the Emulator has been resumed from a paused state (see below). The emulator's disk support code uses both Unix server raw disk device files and regular UFS files to implement DOS file systems.

The machine's VGA display is controlled by code in the Emulator task that directly manipulates the VGA's screen memory. Direct control over the VGA is needed because DOS programs use much more of the VGA's functionality than do any Unix programs, including X.

4.2. Unix Support for DOS File Systems

The DOS emulator uses Unix services to access two different types of DOS file systems: Real DOS file systems residing on the physical hard disk, and the Unix file system as a DOS network file system. A real DOS file system is one laid out using a DOS file system specification. The UFS as a DOS network file system is implemented using the DOS Network Redirector Interface.

4.2.1. Real DOS File Systems

The Real DOS file systems supported by the DOS Emulator are physical DOS file systems laid out on the disk that are pointed to by either a Unix device file or a regular Unix file. Unix device files are used to access resident DOS hard disk partitions or floppy DOS disks. A special device file is set up to point to the resident DOS hard disk partition by the Mach `diskutil` command. The Unix `dd` command is used to copy the image of a DOS floppy into a regular Unix file.

Since the Emulator does not contain a copy of DOS, one of these real DOS file system images must have a version of DOS on it for the Emulator to boot and begin operation. The Emulator will check for both a regular file image with DOS in it or for a special device file at run time. Run time flags can be used to specify boot files other than the defaults.

4.2.2. The UFS as a Network DOS File System

The Emulator implements the Unix file system as a DOS drive using the DOS Network Redirector Interface. Several DOS programs are executed by the V86 thread when DOS boots that initialize

DOS version-specific variables in the Emulator task. Using these variables, the Emulator inserts the fake UFS drive into DOS data structures, making DOS believe that it has an extra disk drive. The Emulator supports this fake DOS disk drive by intercepting the DOS Network Redirector system calls.

There are several benefits to using the UFS as a disk drive under DOS. The large amount of space available under the UFS disk drive provides more room for DOS storage than do the early versions of DOS that limit Real DOS file system disk size. Since the Emulator uses Unix system calls to access files, those files stored directly in the UFS are accessed faster than those stored in the Real DOS file systems which must use the UFS as an intermediary. Finally, the Emulator can easily move data between the UFS and DOS file systems, making file transfers between the operating systems trivial.

The ability to use the UFS as a DOS disk enables a machine that does not have a DOS hard disk partition to effectively utilize the DOS emulator. The Emulator can boot from a floppy image and then use the UFS disk drive as its main drive for execution and storage.

4.3. Unix Support for Managing Multiple DOS Machines

The ability to pause the DOS Emulator and return to a Unix shell enables the user to execute multiple DOS Emulators running in separate DOS "machines." The Emulator can be paused or stopped by pressing the down the Control, Alt, and 'Z' keys simultaneously. By using the Unix server's ability to multitask, the user is able to execute as many DOS programs as he or she wants and to change back and forth between them at will.

The problem of sharing the keyboard and mouse devices between the different DOS Emulators is overcome by using the `select` system call to check for input on the devices before reading them. When a DOS Emulator has been paused and then put into the foreground, the `select` call will return with a `Bad File Descriptor` error because the old file descriptor will no longer have a valid state. This error tells the Emulator to reinitialize the file descriptors for the input device before it continues.

5. Performance issues

The performance of most DOS programs is comparable to that of native DOS machines, in the sense that the Mach DOS system is quite useable as a day-to-day operating environment. In general, the Emulator has been tuned to provide good performance for the average DOS program. Optimizations for specific programs like Windows and Wing Commander were provided when the changes were beneficial to most DOS programs. DOS programs that rely on the fact that privileged instructions execute within one machine instruction period will not perform correctly. Although specific numbers comparing the performance of the Emulator and native DOS have not been calculated, some work has been done to analyze the performance of the Mach kernel's support. This performance is determined by two factors: fault handling overheads, and instruction emulation overheads.

5.1. General fault handling overheads

The fault handling overhead corresponds to the length of time spent in forwarding the V86 thread's exceptions to the appropriate emulation routines. This overhead lies mainly in two areas: the time it takes for the processor to switch back and forth between processor states, and the Mach Kernel fault handling code. Typically, an 80386 processor uses about 500 clock cycles for the switch from V86

mode to Protected mode and back. This roughly translates into about 100 machine instructions. Once the switch from V86 mode to Protected mode has been made, it takes about thirty machine instructions to get to the V86 mode support code in the Mach Kernel. If the V86 exception is handled by the Kernel's emulation code, it typically takes about forty to fifty more instructions to reach the proper code for the type of exception.

5.2. Instruction emulation overheads

The overhead related to the instruction emulation is large compared to native DOS code. The performance of privileged instructions emulated in the Mach Kernel depends on the type of instruction and whether the instruction generates more exceptions or write to the user space. The number of instructions used by the Mach Kernel's V86 support code for each of the following privileged instructions are as follows:

- CLI: 29 instructions to clear the "virtual" interrupt bit
- STI: 37 instructions for the General Protection fault plus another 200 instructions for the trace break point.
- PUSHF: 118 instructions to emulate, 46 instructions for the copyout to user space
- POPF: 121 instructions
- IRET: 131 instructions
- INT, IN, OUT: 21 instructions to leave the kernel V86 support.

Clearly there is a performance cost for programs that use these instructions in tight loops. This cost is one of the reasons why it was decided to move some of the emulation from user space into the kernel.

6. Implementation Lessons

Some lessons learned about Mach during the course of this project were: the suitability of the Mach Kernel's 386 machine dependent code to support a layered OS with intensive machine dependent applications, how to balance the exception handling code between the kernel and user space, how to provide a finer granularity of timing than the kernel supports to a Mach application, and how useful the Mach virtual memory system is for implementing different types of memory support.

Several features were added to the Mach Kernel's 386 Machine dependent code: the ability to take threads into and out of the V86 execution mode, a port-specific protection mechanism for the ISA I/O ports by implementing the bitmap in the task's TSS, new flavors of thread state, and a way to generate external interrupts within the V86 thread.

In order to achieve reasonable execution speeds, some V86 emulation had to be done in the kernel. The types of emulation performed by the kernel are related to hardware specific and non system call instruction emulation. Several different solutions were implemented both within the kernel and user space before the current balance described in this paper was decided upon.

Many DOS programs need time slices smaller than the smallest slice available from the Mach Kernel to perform precise sound output. To provide for this, the Emulator task has to compute the number of timer interrupts required to induce the DOS application into producing the right sounds.

The Mach Virtual Memory system was very successful in implementing the many special features associated with DOS memory and its specifications for memory extension. The Emulator was able to map two different address ranges to the same piece of virtual memory by using an external pager. The Mach VM was very useful for implementing the expanded and extended DOS memory specifications.

7. Conclusions

We have shown that it is possible to implement DOS as a Mach 3.0 application by using a combination of kernel and user level support. As of the writing of this paper, we have implemented DOS as a Mach task that uses the 4.3 BSD Unix server for many of its services. There were problems in this implementation in the form of deficiencies in the Mach Kernel's machine dependent code for the 386 processor which were overcome as described above. Over 100 DOS programs have been effectively run on the Mach DOS support, many of which demand very close integration with the machine hardware. In the future, work will center on severing the Unix dependencies and expanding the functionality of the DOS Emulator even further.

A Causal Distributed Shared Memory Based on External Pagers

F. Boyer
Unité Mixte Bull-IMAG/Systèmes, 2, avenue de Vignate, 38610 Gières, France - Internet: guide@imag.fr - Phone: +33 76634848

Abstract:
The objective of this paper is to propose an efficient implementation of a distributed shared memory based on the Mach External Pager facilities. We concentrate on memories where objects are replicated among nodes, which requires a mechanism for managing the consistency of the copies of an object. Because usual consistency protocols based on strict consistency may lead to a high message traffic in the system, we propose to use a released form of consistency which allows reads and writes to be executed in parallel on copies of a given object. Released consistency allows a significant reduction of the number of messages, which leads to increased concurrency and better performance compared to usual strict protocols. However, as we want to make the implementation transparent to the programmer, some dependency management must be added to the released consistency in order to make the read operations return consistent values with respect to causally related operations. A *causal memory* is a memory respecting the causal dependencies between operations. Such a memory is less consistent than a centralized memory. However, it provides enough consistency for distributed applications in which communication between processes is achieved through shared data.This paper proposes a simple and efficient implementation in which both the released consistency protocol and the causal dependencies management are done by the External Pagers. We show that the cooperative architecture of pagers exactly fulfils causal memory requirements and argue that implementing a causal memory using the External Pagers is very attractive.

1 Introduction

A distributed shared memory is a memory management system which provides a uniform access interface to local and remote objects. While this functionality is very attractive, its implementation may lead to some efficiency issues. The objective of this paper is to propose an efficient implementation of a causal distributed shared memory based on the Mach memory management facilities [1]. A causal memory may is a memory in which read operations return a consistent value with respect to causally related operations. Two aspects must be considered: the consistency protocol that is used to maintain the memory consistency, and the causal dependencies management.

We concentrate on memories in which objects are replicated among the sites. A mechanism for managing object consistency is required. Two main protocols may be used according to the memory consistence they maintain (i.e: strict or released). Strict consistence [2] may be defined as allowing

multiple readers **or** *one writer* to execute in parallel on an object. This protocol provides the programmer with the vision of a global shared memory but may lead to high message traffic. On the other hand, a released form of consistency which allows *multiple readers* **and** *one writer* to execute in parallel on an object, allows more concurrency and increases the system efficiency. We propose to use this released form of consistency for building our causal memory.

However, the released consistency protocol cannot ensure by itself the property of causality. Some dependency management mechanism should be added to the protocol. The management of dependencies requires that read operations return a causally consistent value. Dependency management is the major issue in implementing a causal memory based on a released consistency protocol.

Causal memories have a high potential for increased performance, and are the subject of current research. Different forms of released consistency has recently been introduced, and the Clouds system has proposed algorithms for implementing a causal memory in which multiple writes are allowed to execute in parallel on copies of a given object [3]. A vector timestamp protocol is used to keep the causality relations. Some other works deal with both released and strict consistency protocols, and leave to the programmer the choice of the protocol [4][5]. Munin [6] requires the user to specify the type of its application, which may be difficult for him.

This work has been done in the framework of the Guide project (Grenoble Universities Integrated Distributed Environment) [7]. Guide is a component of Comandos (COnstruction and MANagement of Distributed Open Systems), a project supported by the Commission of European Communities under the ESPRIT program. The aim of Guide is to explore distributing computing, structured in terms of objects and based on a set of heterogeneous workstations. A first prototype of the Guide system has been implemented on top of Unix, and the second version of Guide has been designed on top of Mach 3.0. A preliminary implementation was developed on top of Mach 2.5 [8].

The rest of this paper is organized as follows. Section 2 defines causal memory. Section 3 gives a brief overview of the Guide object-oriented model. The implementation of a causal memory for Guide on top of Mach is described in Section 4. Finally, Section 5 gives some qualitative and quantitative evaluations of the proposed protocol. We conclude in Section 6.

2 Causal Memory

This Section defines what we call a causal memory. Section 2.1 presents the two major forms of consistency: strict consistency and released consistency, and compares their properties with those of a centralized memory. Section 2.2 describes the major issue in implementing a causal memory, that is the management of causal dependencies.

2.1 Strict and Released Consistency

2.1.1 The Strict Consistency Protocol

A consistency protocol is usually based on some readers/writers algorithm which operates on the copies of an object(1). The readers and writers are the sites where the object is accessed (i.e we do not care about individual processes on each site). The strict consistency protocol is based on a form of readers/writers algorithm which follows the rule: **multiple readers or one writer**. Readers are not

(1) Throughout the paper, we use indifferently the term object or page as a set of data which represents the unit of data transfer between pagers and sites.

allowed to execute in parallel with a writer on an object, and writers are exclusive[2]. With such a protocol, a distributed shared memory simulates a centralized one. Indeed, the main property of a centralized memory is that all operations are serialized by the hardware control mechanism. Only one operation is executed at a given time. This property can be kept in a distributed memory, but in this case all the performance expected from distribution is lost. Nevertheless, the absence of a global order on all operation is masked by the strict consistency protocol. Indeed, while there may be several operations in parallel (one per site), the result of any execution is as if all operations were serialized. This is ensured by the property: a read operation on an object returns the value assigned by the most recent write to the object. However, it appears that the strict consistency protocol ensures more consistency than generally needed in distributed environment where the notion of the "most recent write" is not well defined [9]. So the released forms of consistency which allows to have better performance by reducing the level of consistency appear as an interesting research topic.

2.1.2 The Released Consistency Protocol

There are different forms of released consistency. We are interested in those forms which provide a global order on write operations on an object (write operations are not allowed to overlap). Moreover, we want to respect some order on read operations: let us define H as the history of a variable V, $H = (v_1,..,v_n)$. Successive reads yield values with non-decreasing indices in the history. We can summarize the idea of our released consistency by saying that *the effect of a write operation may be delayed.*

The protocol is implemented as a readers/writers algorithm in which readers and one writer are allowed to execute in parallel on different copies of the same object, i.e **multiple readers and one writer**. With such a protocol, two aspects must be considered:

1) Such a memory enforces less consistency than a centralized memory. The property of serialization is not respected (as opposed to the strict consistency protocol). This means that the following situation may arise :

Example (from [9])

Consider two sequential processes P1 and P2 :
P1: Write (O1, 1) ; Read (O2, 0)[3]
P2: Write (O2, 1) ; Read (O1, 0)

Such a situation cannot arise if read and write operations are serialized (at least one of the read operations should return the value 1). Nevertheless, we believe that the absence of serialization has little consequence for a Guide user; in most cases, this will be transparent to him because he does not need to assume the serialization property for implement any algorithm. Indeed, the serialization is often required for low level synchronization algorithms (such as the Dekker Entry Protocol). The Guide user does not need to use such algorithms since high level synchronization facilities are provided to him by the Guide system. Anyway, if the user does assume such a property, the memory management facilities of Mach still allow the use of the strict consistency protocol operating upon specified objects.

---

(2) The netmemory server provided by Mach implements this type of consistency protocol [1].

(3) *read (O, v)* represents a read operation on the object O which returns the value v. In the same way, *write (O, v)* is a write operation which assigns the value v to the object O.

2) Such a protocol does not ensure the causality property. Indeed, we want to insulate the user from the memory implementation issues by providing him a memory which takes into account causal dependencies[4]. The readers/writers algorithm takes care of operations on a single object, and does not consider any causal relations between operations on different objects. Let us take an example to illustrate one of these causal dependencies.

Example (from [3])

Consider two objects O1 and O2 that have 0 for initial value.
Consider two sequential processes P1 and P2:
P1: write (O1,1) ; write (O2, 1)
P2: read(O2 , 1) ; read (O1 , 0)

As P_2 reads the value 1 for O_2, it means that the operation *write (O2,1)* has been executed **before** operation *read (O2,1)*. Thus, *write (O1,1)* has also been executed before *read (O1,0)*, and P_2 should have read the value 1 for O_1. The objective of the dependency management is therefore to take into account the relation *before*. The issue of causal dependencies is explained and formalized in the next Section.

2.1.3 Causal Dependencies

The management of causal dependencies is the most important issue in implementing a causal memory. Let op_i and op_j be two operations related by a causal dependency. We write (op_i -> op_j) if op_i must be executed **before** op_j (in other words, op_j may start only when op_i has finished). We define the dependency relations by three properties:

- Property 1: Let P be a sequential process. P= (op_1 ; .. ; op_n) then (op_1 -> op_2 -> ... -> op_n)
- Property 2: If (op_i -> op_j) and (op_j -> op_k) then (op_i -> op_k)
- Property 3: Let P and P' be two sequential processes.
  P = (op_1 ; .. ; op_n) and P' = (op'_1 ; .. ; op'_m).
  then (op_i -> op'_j) if
  - op_i is a write operation which assigns value v to the variable V at date d, and
  - op'_j is a read operation on the variable V, which returns the value v at date d', $d<d'$.

The dependencies induced by Property 1 correspond to the sequential aspect of a process. We call these dependencies **intra-process dependencies**. In the same way, the dependencies induced by Property 3 correspond to inter-process communication and are called **inter-process dependencies**.

In the previous example, we have the following dependencies: (write(O1,1) -> write(O2,1)) and (read(O2,1) -> read(O1,0)) by Property 1 ; (write(O2,1) -> read(O2,1)) by property 3. Then (write(O1,1) -> read(O1,0)) is generated by Property 2 and implies that the read operation on O1 returns the value 1 instead of the value 0.

2.1.4 Summary

A *causal memory* is a memory which preserves the causal dependencies between operations (as defined in 2.1.3). The operations executed by remote processes sharing data are partially ordered.

(4) Demonstrating that causal memories are appropriate for programmers is out of scope of this paper, however many papers have shown this appropriateness [3] [9].

A causal memory manages this partial order. Thus, causal memories are useful for distributed applications in which communication between remote processes is achieved through shared data. The proposed solution is based on a released consistency protocol, completed by the management of both forms of causal dependencies (i.e intra-process dependency and inter-process dependency). Our released consistency protocol does not allow multiple writers to execute in parallel on an object. This exclusion of writers is in fact not required for implementing a causal memory. However, allowing multiple writers to execute in parallel would require the management of the versions of objects. Because the causal memory is implemented in the pagers, exclusive writers allows to implement a causal memory in a very simple and efficient manner (without managing versions of objects). Thus, the released consistency protocol that is used only allows multiple readers and one writer to execute in parallel on an object.

3 Overview of the Guide Object-Oriented Model

While the proposed memory is somehow independent from the Guide system which is implemented on top of Mach, we describe the main feature of the Guide model in order (1) to situate the context of the work and (2) to prove the correctness of the proposed implementation. The Guide model includes the followings aspects :

- An **object** model: objects provide a convenient means for application structuring. Objects are persistent, i.e. their lifetime is not related to that of the execution unit in which they were created. Within the system, they are designated by low-level, location-independent identifiers. The object model is accessible to the users through a specific language whose run-time system is implemented on top of the Guide kernel.
- A computational model, which provides the user with the concept of a **job**. A job represents a virtual machine, which hides the details of distribution, and provides mechanisms for concurrency control. A job groups together in a common address space a set of objects and a set of threads of control, called **activities,** that operate upon these objects. Communication between activities belonging or not to the same job is achieved through **shared objects**. For example, in figure 1, the two jobs *Job1* and *Job2* share the objects *O3* and *O4*.

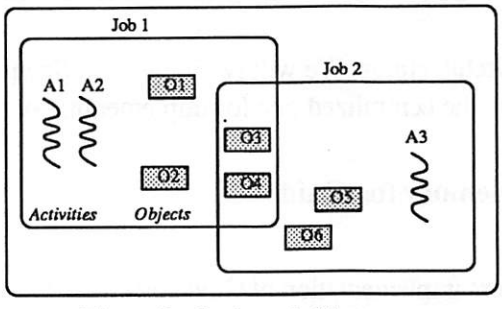


Figure 1: sharing of objects

Guide also provides the **distribution of jobs**: jobs may be distributed on a set of sites. A diffusion mechanism allows a job to dynamically extend to remote sites. Thus, a job may be seen as a set of representatives, also called local jobs, one on each site to which it has diffused. Figure 2 illustrates distribution for two jobs. Job1 is entirely represented on site Node1, whereas Job2 is composed of two local jobs Job 2/Node1 and Job 2/Node2.

Thus, the main notions that must be considered for our proposal deal with the concepts of jobs, activities, shared objects and distribution of jobs.

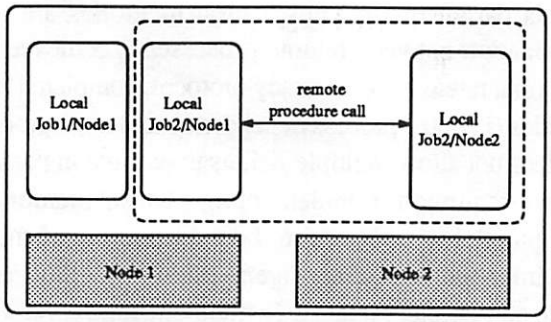


Figure 2: distribution of jobs

4 Causal Memory for Guide on Top of Mach

The objective of this Section is to describe an implementation of a causal memory for Guide on top of Mach. Section 4.1 recalls the principles of the Mach memory management facilities and Section 4.2 proposes a realization of our causal memory on top of Mach.

4.1 Memory Management Facilities on Top of Mach

The Mach kernel allows the user to provide paging services outside the kernel. This is based on external servers (or pagers). A pager allows to create memory objects (i.e chunks of memory), identified by ports[5]. To address a memory object, a thread maps it in the virtual address space of its task. Once an object is mapped, page faults on this object are processed as usual page faults. The kernel sends a page fault to the port which identifies the object. This allows to implement two main architectures of pagers:

- A centralized architecture, in which an object is managed by a unique server. Paging may be remote.
- A cooperative architecture, in which pagers cooperate to maintain object consistency. Paging is always local: a kernel always sends a page fault to its local pager.

Figure 3 illustrates these two architectures. We will see in the next Sections that the cooperative architecture is more adequate than the centralized one for implementing our causal memory.

4.2 Proposal for a Causal Memory for Guide

4.2.1 Guide on Top of Mach 2.5

Let us briefly describe the current implementation of Guide on top of Mach in order to specify the interaction between the Guide concepts and the pagers. Mach provides the concept of a task which is a multi-threaded address space. Thus, each *local job* is implemented by a task and each *local activity* is implemented by a thread. Object sharing between activities in the same local job is implicit because the threads implementing these activities share the same address space. Object sharing between activities in different local jobs is implemented using pagers which allow object sharing between different tasks on the same sites or on remote sites. Before invoking a Guide object, a local

(5) A port identifies a unique Mach object in the network.

(6) Throughout the paper, we use indifferently the term object or page as a set of data which represents the unit of data transfer between pagers and sites.

job must bind the object in its context. This binding operation maps the object into the address space of the task which represents the local job.

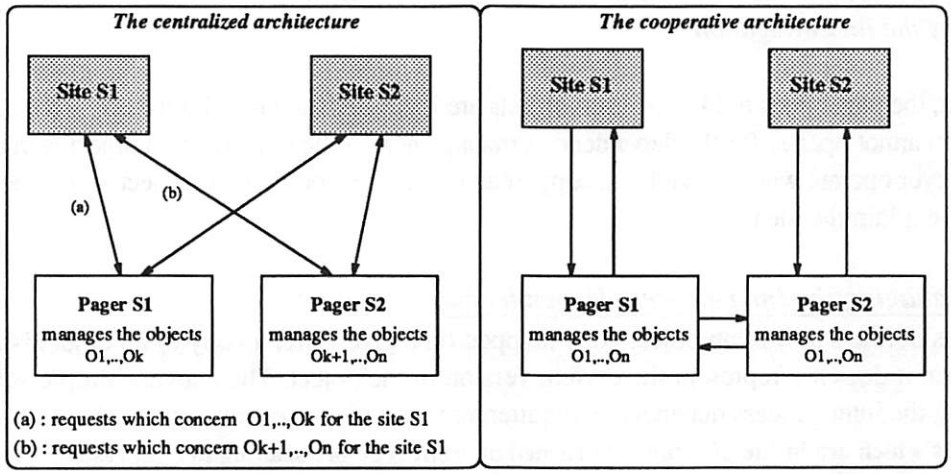


Figure 3: Centralized and cooperative organization of pagers

4.2.2 Proposal

Our proposal consists of implementing a causal memory based on a released consistency protocol. The released consistency protocol is directly enforced by pagers. Pagers apply the rule *one writer and multiple readers* on objects (i.e readers are not invalidated when there is a writer). As a consequence, readers are not always reading the last version of objects. The management of dependencies is therefore in charge of specifying the latest moment when readers should be invalidated so as to keep causal dependencies valid.

While the dependency management can be easily implemented in the object-invocation mechanism [3], we have choosen to implement it in pagers. Indeed, implementing efficiently the dependency management in the object-invocation mechanism requires the knowledge of the behavior of any method, in the sense that we must be able to determine whether or not a method modifies the state of an object. Such a criterion may lead to consider any method which potentially may write as a writing method. The expected gain is consequently reduced since the efficiency of shared memory relies on the fact that the number of read method is significant. Moreover, bracketing method invocations with calls to the consistency protocol is cumbersome, especially if methods have short execution times.

Thus, as pagers have the information which allows them to determine the types of potential accesses (read/write) to each object, we propose to use this information to manage the causal dependencies. This provides the additional advantage of implementing a stand alone causal memory. The dependency management is explained below.

4.2.3 Management of the Causal Dependencies

For more simplicity, we consider, in a first step, that there is a unique pager in the network. Distributed pagers are considered in Section 4.2.4. Pagers have to manage both intra-process and inter-process dependencies, as defined in Section 2.1.3. In the context of Guide, the management of dependencies requires the following properties:

Inter-process dependencies requirement: if an activity invokes successively two shared objects O1 and O2, it will work on consistent copies of O1 and O2. In other words, if O2 has been

assigned a value vO2 before O1 has been assigned a value vO1, then if the activity accesses the value vO1 for O1, it must accesses a value for O2 at least as recent as vO2.

Intra-process dependencies requirement: if an activity successively invokes the same object O, then the second invocation will work on a copy of O which is at least as recent as the copy of O used by the first invocation.

In fact, the pager does not know when objects are invocated, and by which activity they are. Thus, the pager cannot operate for the dependencies management at the same time an object is invocated. It can however operate when providing a copy of an object in response to an object fault. The two next Sections explain the idea.

A) Managing the Inter-process Dependencies

Let us define states of an object copy mapped on a given site: a copy of an object is in the **old** state when it does not represent the current version of the object. Then, a very simple solution for satisfying the inter-process dependency requirement (as defined previously) consists in **invalidating all objects which are in the old state and cached on a site S before sending to S (in response to an object fault) an up-to-date version of an object which was previously in the old state.** There may be fault requesting an object which was previously in the *old* state either when the *old* object has been invalidated, or when the involved kernel requests write rights on the object (indeed, an object in the *old* state can only be accessed in read mode). As only the faults requesting an object which was previously in the *old* state require some invalidations of other *old* objects, the faults requesting an object which is not shared between sites do not involve invalidations (since only the objects that are shared may become *old*). Notice that there can be no more invalidations than with the usual strict consistency protocol since, as soon as an object becomes *old*, some invalidations are avoided.

So, when sending an up-to-date version of an *old* object O on a site S, the pager previously invalidates all copies of shared objects which are not as recent as the copy of O which will be sent. This ensure that an activity cannot access successively two copies of objects O1 and O2 such as the accessed copy of O2 is older compared to the accessed copy of O1. While this solution is convenient due to its simplicity, we propose to use the Mach concept of task to improve it by reducing the number of invalidations of *old* objects. Indeed, the pager knows, for each object, in which tasks it is mapped. Let us define a relation *Shares (T, T', S)* which related two tasks *T* and *T'* which are present on a site *S*. The relation *Shares* is defined as follow:

. If T and T' map a common object on the site S, then Shares(T, T', S).

. If Shares (T, T', S) and Shares (T', T'', S) then Shares (T, T'', S).

. If Shares (T, T', S) then Shares (T', T, S).

The relation *Shares* defines a set of tasks which are related through shared objects (in a direct or indirect manner).Then, **when the pager sends an up-to-date version of an object O in response to an object fault, the set of old objects which have to be invalidated is reduced to old objects which (1) are mapped in tasks which map O or (2) which are in relation Shares with tasks which map O.** The following example illustrates this idea.

Example

T1 is a task on site S, and T2, T3, T4 are three tasks on S'. Objects O1,..,O4 have 0 for initial value.
Suppose the following execution:
Task T1, Site S: **write (O1, 1)** *; write (O2, 1)* .....*O1 and O2 becomes old on S'*

Task T2, Site S' : *read (O2, 1) ; write (O3, 1)*
Task T3, Site S' : *read (O3, 1) ; write (O4, 1)*
Task T4, Site S' : *read (O4, 1) ;* **read (O1, 1)**

In fact, T4 could not read another value than 1 for O1 since there are the relations (write(O1, 1) -> write(O2, 1) -> read(O2, 1)-> ... -> read(O1, 1)). But, if T2 reads the value 1 for O2, it means that an up-to-date version of O2 has been sent to S'. Thus, every old objects on S' which may be accessed by tasks which are in relation Shares (that is T3 and T4) with the tasks which have mapped O2 (that is T2) have been invalidated before the sending of the up-to-date version of O2. Therefore, O1 has been invalidated and the copy accessed by T4 is at least as recent as the accessed copy of O2 by T2.

Thus, we propose two solutions for inter-process dependencies management: a simple one which invalidates all *old* objects of a site S before sending an up-to-date version of an object to S, and a more complex one which allows to reduce the number of *old* object invalidations by using the concept of task. However, performance of this second solution will have to be compared with the simplest one. Indeed, the cost of the use of information about tasks (i.e the management of the relation *Shares*) may be too expensive, leading to adopt the previous and simplest solution.

B) Managing the Intra-process Dependencies

If the management of intra-process dependencies is straightforward when processes do not migrate (or diffuse) to several sites, it leads to some difficulties when a process is allowed to execute on a set of sites. Process migration is very useful for fault-tolerance or load-balancing. Assuming that a process executes on only one site would be too restrictive. Thus, we have choosen to take into account the case of processes which are allowed to migrate, in such a way that it does not cost when they do not migrate. In Guide, because of the sequential character of an activity, we must satisfy the following property, as illustrated in the next example:

Example

Activity A1 on site S1:
Invocation(O)
A1 diffuse on S2 .....> Activity A1 on site S2 :
Invocation(O)

Due to the sequential aspect of an activity, the copy of O accessed on site S2 must be at least as recent as the copy of O accessed on S1. Thus, an inconsistency may arrive if the copy of O on site S1 represents the up-to-date copy of O, and the copy of O on S2 is an old copy of O.

To solve this problem, the intra-process dependencies are managed at the instant an activity migrates from one site to another. The principle is the following: **when an activity A migrates from site S to site S', let T be the task of the thread which represents A on S', then the pager invalidates either every object O which is in the old state on S' (this is the simple solution), or each object O which is in the old state on S' and which is mapped in T or which is in relation Shares with T (this is the more complex solution which allows to reduce the number of invalidations as defined previously in 4.2.3.A).** The reader may think that this solution is not convenient since each diffusion requires the emission of a synchronous message to the pager. Nevertheless, if some objects must be invalidated, then the cost of the message is balanced by the fact that the invalidations of such objects have been delayed to the latest moment. Moreover, we will see in the next Section that this communication

between the activity and the pager is local. This allows to locally share information between the activities and the pager, so that an activity sends a message to the pager only when there are objects to invalidate. Note that we do not need to synchronize the access to this information since it is only modified by the pager.

4.2.4 Taking Distribution of Pagers into Account

Because of the distribution of pagers, the required information (like the list of *old* objects on a given site) is distributed. In Section 4.1, we have presented two possible architectures for the pagers: a centralized one and a cooperative one. While the current implementation of the Guide prototype uses the centralized architecture, we intend to switch to a cooperative architecture for implementing the proposed causal memory. Indeed, there are two advantages of this decision:

- First, it allows to have local paging, instead of remote paging (as with the centralized architecture).
- Second, with a cooperative architecture, information is distributed in an adequate manner for the protocol requirements. Indeed, each pager knows every thing about the objects that are cached into the site it manages (like the mapping of objects to the tasks of its site). This information is sufficient to manage the dependencies.

Pagers have to cooperate in order to ensure that there is no more than one writer on a given object at a specified time. For this, we associate an *owner pager* to each object. The owner pager of an object is in fact the pager of the site which has the latest version of the object. While any kernel sends an object fault to its local pager, the receiving pager may forward the request to the owner pager of the given object. If the owner migrates from one site to another, a forward link allows to find it.

Finally, the overall execution is described as following. An object which is mapped on a site can have four states as seen by the local pager:

- **read**: the site has *read* rights on the object, and has the latest version of the object.
- **write**: the site has *write* rights on the object, and has the latest version of the object.
- **old**: the site has *read* rights on the object, and has an old version of the object.
- **invalid** the object has been invalidated, which means that it is conceptually not more present on the site.

In addition, a pager manages a set of attributes for each object:

- **owner**: gives the identification of the owner pager.
- **page_out**: true if the object has been paged out from the local kernel. When receiving a page fault request from the local kernel, this information allows to know whether the page fault has been caused because the accessed page was paged out. In this case, the pager has simply to provide the kernel with the paged out copy of the page.
- **readers_list**: lists the pagers of the sites which have the object in reading mode. This allows the owner of a page to notify all readers when the page is accessed in write mode.
- **tasks_list** : lists the tasks of the site which map the object.

For each task which is represented on the current site, the pager manages a relation **Shares**. *Shares (T)* gives the list of the tasks which are related with task T through shared objects (as defined previously). The relations *tasks_list* and *Shares* allows, when sending an update version of a shared object to the local kernel, to determinate the tasks for which *old* objects must be invalidated. As previously said, a more simple solution allows to avoid the management of the relation *Shares* but

leads to more invalidations: every *old* objects that are on the local site are invalidated when sending an up-to-date version of a shared object to the local kernel.

Because the entire protocol is difficult to describe, we have choosen to illustrate the protocol by simplified pseudo-algorithms. Detailed algorithms are given in Annexe A.

1) When receiving an object fault for an object O:

If O has just been paged out, and is not is the invalid state, the pager provides the requesting kernel with its local copy of O and returns.

If the pager is not the owner of O, it communicates with the pager owner of O in order to get an up to date copy of O. The owner of O may change regarding to the requesting mode for O:

If O is requested in write mode then the local pager becomes the owner of O and O turns in the old state with read rights only on the other sites where it is mapped.

Then, before sending the copy of O, the pager invalidates:

. *with the simple solution: every old object on the current site*

. *with the more complex solution: the old objects on the current site which (1) are mapped in tasks which map O or (2) which are mapped in tasks in relation Shares with the tasks in which O is mapped.*

2) when receiving a notification of an activity migration:

Let T be the task of the thread which represents the activity on the destination site.

The pager of the destination site invalidates:

. *with the simple solution: every old objects on the local site*

. *with the more complex solution: the old objects which are mapped in T, or which are mapped in tasks which are in relation Shares with T.*

5 Evaluation

5.1 Adequacy of Mach for the Protocol

We can make the following remarks regarding the adequacy of Mach for the proposed protocol:

First, due to the fact that Mach allows the user to provide paging services outside the kernel, we had two possibilities: implementing our causal memory in the object-invocation mechanism or in the pagers. We have shown that the possibility of managing dependencies in the pagers is very attractive. Indeed, it avoids bracketing method invocations with calls to the consistency protocol without having an efficiency decrease.

Secondly, the External Pager facilities also allow to implement a specific architecture of pagers. We have shown that the cooperative architecture completely satisfies our causal memory requirements. Moreover, it allows local paging. In addition, since each object can be treated in an independent manner, it allows to use both a strict consistency protocol for specified objects (which require the serialization property for example), and a released consistency protocol for other objects.

Finally, the invalidation mechanism avoids to refresh copies of objects which are no more accessed. Nevertheless, the kernel paging interface does not allow to request more than one object invalidation per message. As the proposed protocol often requires the invalidation of a set of objects

at the same time, the ability for a pager to request more than one object invalidation in a message would be useful for our protocol.

5.2 Some Quantitative Evaluations

The following figures compare the cost of the released consistency protocol implemented by the cooperative architecture of pagers with the usual protocols, which are strict consistency protocols implemented either by the centralized or by the cooperative architecture of pagers. Comparisons are given in terms of the number of exchanged messages.

Figure 5 (on next page) presents an evaluation in a favorable case, where a page is both accessed in the write mode on site S1, and in the read mode on another site S2. Recall that, with the strict consistency protocol, there cannot be parallel read and write accesses on the given page on sites S1 and S2. Thus, we suppose that, in the case of the strict consistency protocol, each of these accesses causes a page fault.

In fact, consequences due to the dependency management are not represented in this evaluation. Remember that the effects of the dependency management is to invalidate pages in order to prevent inconsistencies. Then, even if the dependency management leads to an invalidation of the accessed page on the site S2 (which has an old version of the page since the page is on S1 in write mode) at the time of the S2's second access (namely, *d)read*), the proposed protocol is still more efficient than the strict consistency protocol (see the case n=2).

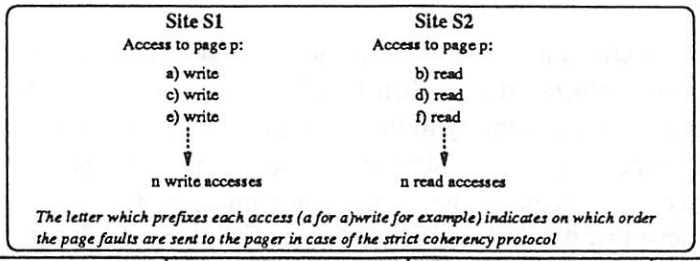
Figure 6 presents an evaluation in an unfavorable case, where a page is both accessed in the write mode on site S1 and on site S2.

The reader can see that the cost of the proposed protocol is in fact the same as the cost of the strict consistency protocol implemented by the cooperative architecture of pagers. When compared with the centralized architecture of pagers, the cost of the proposed protocol seems to be acceptable: if the number of messages may be greater, the number of remote communications is smaller.

5.3 Evaluation Conclusion

As the proposed protocol has not been yet implemented, we do not give measurements in this paper. Nevertheless, we believe that the previous evaluations show the gain that we could expect from such a causal memory because they reflect the real number of exchanged messages.

Anyway, we can remark that the gain is due to the delayed of invalidations of pages, which leads to both a **decrease of the number of page transfers and a decrease of the number of invalidations** compared with a strict consistency protocol. To be fair, it should be stated that, although the number of transfers and invalidations is reduced, there is more time spent to manipulate the data structures like *tasks_list* , or others. But this CPU time is negligible compared to the gain we get when reducing the number of remote communications. Moreover, we proposed two solutions for the dependency management (see Section 4.2.3): a very simple one which implies more invalidations than the second, but which is still more efficient than the usual strict consistency protocol. The second solution is more difficult to evaluate: it requires less invalidations but may lead to the management of somewhat complex data structures.



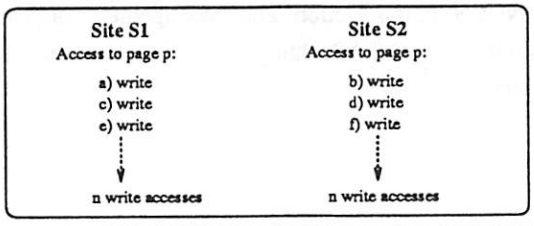
The letter which prefixes each access (a for a)write for example) indicates on which order the page faults are sent to the pager in case of the strict coherency protocol

	Strict consistency + Centralized arch. of pagers	Strict consistency + Cooperative arch. of pagers	Released consistency + Cooperative arch. of pagers
Number of page faults	2n	2n	n
Number of messages	8n–2	12n – 4	12
Number of remote messages	4n–2 to 8n–2	4n – 2	4
Number of page transfers	3n	3n + 1	4
Numb. of remote page transfers	n to 3n	n	1

Case n = 2

Number of page faults	4	4	2
Number of messages	14	20	12
Number of remote messages	6 to 14	6	4
Number of page transfers	6	7	4
Numb. of remote page transfersss	2 to 6	2	1

Figure 5: evaluation in a favorable case



	Strict Consistency + Centralized arch. of pagers	Strict Consistency + Cooperative arch. of pagers	Released consistency + Cooperative arch. of pagers
Number of page faults	2n	2n	2n
Number of messages	8n–2	12n–4	12n–4
Number of remote messages	4n–2 to 8n–2	4n–2	4n–2
Number of page transfers	4n–1	6n–2	6n–2
Numb. of remote page transfers	2n–1 to 4n–1	2n–1	2n–1

Case n=2

Number of page faults	4	4	4
Number of messages	14	20	20
Number of remote messages	6 to 14	6	6
Number of page transfers	7	10	10
Numb. of remote page transfers	3 to 7	3	3

Figure 6: evaluation in a unfavorable case

6 Conclusion

In this paper, we have described a simple mechanism for implementing an efficient distributed shared memory for the Guide object-oriented distributed system on top of the Mach kernel. The proposed memory is called a causal memory in the sense that it respects the causal dependencies between operations. Such a memory does not simulate a centralized one because it does not provide the serialization property. Nevertheless, the serialization property is not required by most applications, and we believe that it will be sufficient for the Guide users. Moreover, the proposed mechanism can be used in parallel with the usual strict consistency protocol. This allows to use a strict consistency scheme only when needed.

Our proposal is innovative since there are few systems which implement such a memory. Causal memories just start to become a subject of investigation, and most of the relevant papers only present a theoretical point of view. Thus, we believe that our proposal is interesting because we describe a simple and efficient implementation of such a memory by using the Mach memory facilities. The mechanism seems to be efficient, since the number of communications (and especially the number of page faults) is reduced compared to the usual protocol for the consistency management (based on a strict consistency scheme). Finally, as our proposal is based on the External Pagers facilities provided by Mach, we believe that the proposed memory could be used by any system which is implemented on top of Mach.

Acknowledgements

I would like to thank Professor Sacha Krakowiak and Xavier Rousset de Pina for their help in reviewing this paper.

The Guide project is supported by the commission of European Communities through the ESPRIT program in project COMANDOS (Construction and Management of Distributed Open System), the Universities of Grenoble (Institut National Polytechnique de Grenoble - Université Joseph Fourier) and Centre National de la Recherche Scientifique.

Annex A
Algorithms

```
PROC Pager_receives_request_for_page_from_kernel (page, mode)
[ This request is executed by a pager when it receives a request for a page fault from the local kernel]
 if (page_out [page]) and (not state[page]=invalid) then return the page to the requesting kernel
 else if  owner [page] = myself
  /* this means that the requested mode is write when the current mode of the page is read */
                for each pager P in readers_list [page]
                do   Pager_ask_for_page_to_become_old (P, page)
                list_readers [page] = Null ; state [page] = write
                return the page to the requesting kernel
          else  /* I am not the owner */
             if (state [page] = read )
                    Pager_ask_for_rights_to_owner ( owner [page], page )
                   return the rights to the requesting kernel
              else   Pager_ask_for_page_to_owner (owner [page], page, mode)
                   statee [page]  =  mode
                    if (state = write ) owner [page] = myself ;  readers_list [page] = NULL
                    /* because the requesting kernel will receive a new version of the page,
                     we must manage the inter-process dependencies */
                  for  each task T in  tasks_list[page]
                    for   each task T' in  Shares[T]
                     for    each page p such as (mode[p] = old) and (T' in tasks_list[T])
                       do  state [p] = invalidate ;  invalidate (p)
                  return the page to the requesting kernel
 EndProc

 PROC  Owner__receives_request_for_page_from_pager(page,  mode)
 [ This procedure is executed by a pager which is (or has been) the owner of the  given page. This procedure
replies to the request   Pager_ask_for_page_to_owner, coming from another pager ]
      if (owner [page] <> myself ) forward_to (owner [page] )
      else if  (mode = write)
                owner [page] = asking_pager ; mode [page] = old
                 for each pager P in readers_list[page]
                 do  Pager_ask_for_page_to_become_old(P,  page)
          else   readers_list [page] += asking_pager ; state [page] = read
               /* in fact,  we choose to reduce the rights of the current site, because it is
              more probable that the requesting site will soon access the page in write mode */
          return the page to the requesting pager
 EndProc
```

```
PROC  Owner_receives_request_for_rights_from_pager (page)
```
[This procedure is executed by a pager which is (or has been) the owner of the given page. This procedure replies to the request Pager_ask_for_rights_to_owner, coming from another pager]

/ as the requesting rights are necessary in write mode, this procedure executes as the previous procedure, in the case (mode = write) */*

```
EndProc

PROC  Pager_receives_order_for_page_to_become__old (page)
```
[This procedure is executed by a pager, when it receives the request Pager_ask_for_page_to_become_old from another pager]

```
    state [page] = old
EndProc

PROC  Pager_receives_request_for_map_from_task (task, page)
```
[This procedure is executed by a pager when it receives a request for binding an object from a local job]

```
    Shares [task] += tasks_list [page]
    tasks_list [page] += job
    if (state [page] = old )
            state [page] = invalidate
             if (page_out [page]) page_out [page] = false
             else invalidate (page)
EndProc

PROC Pager_receives_notification_of_job_extension(job, task)
```
[This procedure is executed when an activity migrates from one site to another (assume S'), and when there is objects to invalidate. It is the local pager (on S') which executes this procedure]

```
  for each task T  in  Shares[task]
    for each page P such as (T in tasks_list [P] or ) and (mode [P] = old)
    do  invalidate (P) ; mode [P] = invalidate
EndProc
```

Bibliography

[1] Alessandro Forin, Joseph Barrera, Richard Sanzi. The Shared Memory Server. *Usenix*, winter 1989.

[2] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9), September 1979.

[3] M. Ahamad, P. W. Hutto, R. John. Implementing and Programming Causal Distributed Shared Memory. *11th International Conference on Distributed Computing Systems*, May 1991.

[4] W. K. Giloi, C. Hastedt, F. Schoen, W. Schroeder-Preikschat. A distributed implementation of shared virtual memory with strong and weak coherence. *Distributed Memory Computing, 2nd European Conference*, EDMCC2, April 1991.

[5] P. K. Sinha, H. Ashihara, K. Shimizu, M. Maeckawa. Flexible User-Definable Coherence Scheme in Distributed Shared Memory of Galaxy. *Distributed Memory Computing, 2nd European Conference*, EDMCC2, April 1991.

[6] Bennett, J. K. , Carter, J. B. and Zwaenepoel, W. . Munin: Distributed Shared Memory based on Type-Specific Memory Coherence. *Proc 2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 168-175, 1990.

[7] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, G. Vandôme. Architecture and Implementation of Guide, an object-oriented distributed system. *Usenix Computing Systems*, 4(1) Winter 1990.

[8] F. Boyer, J. Cayuela, P.Y. Chevalier, A. Freyssinet, D. Hagimont. Supporting an object-oriented distributed system: experience with Unix, Mach and Chorus. *Proceedings of the 2nd Symposium on Experience with Distributed and Multiprocessor Systems*, Atlanta, Mars 1990.

[9] P. W. Hutto, M. Ahamad. Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories. *10th ICDS*, 1990

Supporting Structured Shared Virtual Memory under Mach

Ray Bryant
Paul Carini
Hung-Yang Chang
Bryan Rosenburg

IBM Research Division
Thomas J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598
{raybry,carini,hychang,rosnbrg} @ watson.ibm.com

Abstract

Distributed Shared Virtual Memory (or DSVM) has been proposed by several researchers as a method for simulating shared memory on a loosely coupled multicomputer. The principle advantage of the DSVM approach is that it provides a shared memory programming model that is familiar to many users, and this simplifies the migration of applications from conventional machines to the loosely coupled environment. However, for use in high-performance parallel processing, the standard DSVM model has several fundamental disadvantages that keep it from performing as well as traditional message-passing models of parallel computation. In this paper, we present a new version of shared virtual memory, called Structured Shared Virtual Memory or SSVM, that overcomes these difficulties. We discuss the implementation of SSVM under the Mach operating system and we present preliminary performance comparisons between this implementation of SSVM and the standard implementation of DSVM under Mach.

Introduction

Shared-memory multiprocessors present a natural programming model to the user, but it is commonly accepted that it is more expensive to build such machines than it is to connect networks of individual machines. The low cost of such networks of machines has led to the rise in popularity of the message-passing paradigm for parallel processing. However, the message-passing model has the disadvantage that conversion of programs from uni-processors or shared-memory multiprocessors can be difficult and time-consuming.

In the distributed programming domain, several researchers have proposed using a *distributed shared virtual memory* or DSVM to provide a shared memory programming paradigm on a network of machines that do not share physical memory. (See, for example, [18][13][6][14][17][20] . See also the survey articles [21][19] .) One rationale for the DSVM model of computation is that it can simplify program development for the distributed environment; it can also simplify the process of migrating programs from uniprocessor to distributed environments. As the speed of communication networks increase, the distinction between distributed and parallel processing begins to blur. It is therefore natural to consider the use of DSVM for high performance parallel processing.

However, high-performance parallel applications written to run on a shared-memory multiprocessor do not, in general, run efficiently in the DSVM environment. Nor are programs written for the latter environment as efficient as carefully balanced message-passing programs for the same hardware. These problems are caused, in part, by the mismatch between the sizes of shared objects and the size of the virtual memory page frame, and by the fact that all data transfer in a DSVM system is demand-driven, limiting the potential for overlapping data transfer with computation.

To solve these problems we present in this paper a new DSVM design, called Structured Shared Virtual Memory or SSVM, and discuss the implementation of SSVM under the Mach operating

system. SSVM overcomes the problems of DSVM for parallel processing by letting the user explicitly define the partitioning of shared memory into subregions that are meaningful to the application, and by letting the user specify the points at which particular objects should be pushed to other machines. Using this information, SSVM can transfer data among processors more efficiently than would be possible under a DSVM system.

In the following sections, we first discuss what we perceive to be the fundamental limitations of traditional DSVM environments for high-performance parallel processing. We then define the SSVM interface and discuss two possible implementation strategies, both based on the Mach [1] operating system. Next, we present the status of our SSVM prototype and initial performance measurements on a system consisting of a network of IBM Risc System/6000 workstations. We compare the performance of the SSVM system with that of the Netmemory server distributed with Mach 2.5 [14]. We then evaluate the appropriateness of the Mach system interface for the implementation of SSVM, and we discuss extensions to the interface that would simplify our implementation or make it more efficient.

Distributed Shared Virtual Memory

For our purposes, a Distributed Shared Virtual Memory (or DSVM) system is a system such as Ivy [18] that provides a simulated shared memory region to a collection of application processes running on machines that do not share real storage. Any process can address any memory location in the DSVM region, and except for performance, the fact that the storage is not really shared is invisible to the application processes that use it. A DSVM system depends on the virtual memory mapping hardware of the underlying machines to detect memory accesses that might result in inconsistent views of the shared region. The process or processes making such accesses can be delayed while the system moves data among machines to provide a consistent picture. A DSVM system can be implemented on top of a native operating system if the underlying system provides necessary facilities for manipulating the virtual memory mapping hardware. Otherwise the DSVM system must be implemented as part of the operating system itself.

A DSVM system uses a coherency protocol to maintain data consistency among machines. Most DSVM implementations use a write-invalidate coherency protocol [19]. Under this protocol, a page in the DSVM region is in one of two states: *read-only* or *writeable*. There can be copies of a read-only page on many processors of the DSVM system. A writeable page is only present on one processor. The DSVM software depends on the virtual memory hardware and on the operating system virtual memory manager to signal the DSVM system when an application process attempts to write a read-only page or attempts to read a writeable page resident on some other processor. When an attempt to write a read-only page is detected, the DSVM software causes all other copies of the page to be discarded ("invalidated") and the page to be converted to a *writeable* page resident on the processor that detected the write operation. When an application process accesses a writeable page resident on another processor, the DSVM system converts the page to a read-only page and copies it to the requesting processor.

DSVM systems using the write-invalidate protocol can present to their users a functional equivalent of shared memory [18][21] . Indeed the write-invalidate protocol is inspired by similar protocols used to manage the processor caches on shared real-memory multiprocessors. However, a DSVM system cannot provide shared memory that has performance equivalent to that provided on a shared real-memory multiprocessor, because

- The granularity of sharing is much larger (page versus cache line) in the DSVM case.
- Inter-processor transfer times are 3-4 orders of magnitude larger in a DSVM system than in a shared real-memory multiprocessor.
- Software overheads for coherency protocol management are much larger than the corresponding hardware overheads for cache coherency.

The fact that the unit of sharing is the entire page in a DSVM system implies that the false sharing problem is much worse in such systems. False sharing occurs when different processors access

different data objects that *happen* to be resident in the same page. The page appears to be shared to the DSVM system even though the original objects were not shared. If the processors attempt to update the objects at the same time, the page can be bounced back and forth between the the processors multiple times before the updates can complete. While this "ping-pong" effect is also observed in the cache consistency case, it does not occur as frequently because cache lines are smaller than pages and are therefore less likely to hold more than one object. Furthermore, the time required for cache-line transfer is infinitesimal compared to a DSVM page transfer. The result is that the performance impact of false sharing in a DSVM system can be severe.

Of course, page thrashing (or cache-line thrashing) can also occur because of true data sharing. Certain hardware versions of DSVM systems (e. g. Plus [7]) solve the page thrashing problem by using a write-update coherency protocol in which updates to the read-only copies of a page are applied every time the page is changed. The write-update protocol is infeasible in software DSVM systems since there is no efficient way to cause the updated data to be copied across the network on an update by update basis.

Researchers have tried a variety of techniques that attempt to alleviate the page thrashing problem. For example, in Mirage [13] a page remains writeable on a requesting processor for a minimum time interval, and requests for the page by other processors are delayed until this time interval expires. Similarly, in Platinum [10], a NUMA-class shared-memory multiprocessor operating system that is in some ways similar to a DSVM system, a page is "frozen" if the system determines that the page is thrashing between processors. Munin [6] uses type-specific memory coherence; the programmer associates a type (write-once, read-mostly, etc.) with shared data and different coherency protocols are used to manage data of different types. To avoid page thrashing due to false sharing, a programmer can specify the type as *write-many* in which case the system uses a delayed-write policy. Under this policy, Munin does not guarantee a precise simulation of shared memory semantics.

Our approach to solving the page thrashing problem is to adopt a memory coherency model similar to, but more aggressive than, the weak consistency model of memory coherency for shared memory multiprocessors [12][16][19] . The weak consistency model does not require that processors see a consistent view of storage at the completion of each store operation. Instead, the model only requires storage to be consistent after synchronization points. The argument for weak consistency is based on the assumption that correct parallel programs do not access shared memory in an undisciplined manner, but instead use synchronization primitives to ensure that a consistent application-level view of the shared storage is maintained at all times. Thus it is not necessary to keep storage completely consistent between synchronization points because the programmer has already ensured that a processor will not examine storage that is simultaneously being changed by another processor. Programs that do not maintain this discipline are, by definition, faulty. (For the purposes of this paper we ignore such applications as chaotic relaxation that execute properly in spite of concurrent read and write accesses to shared storage.)

We make an analogous argument for DSVM systems: It is not necessary to keep DSVM storage completely consistent at all times. It is enough to present a consistent view of storage at certain points of a DSVM program's execution. A page that is falsely shared because it contains multiple objects can be inconsistent while different processors update the objects, so long as the updates to any particular object are propagated to other processors before the object is "released", where the notion of object release is specific to the programming model of the particular application. The problem, of course, is that without help from the application itself, the DSVM system cannot determine the object boundaries within a page nor can it determine the points at which the updates to an object must be propagated. Structured Shared Virtual Memory includes facilities through which an application can provide this information to the system.

For parallel processing, DSVM systems have the additional disadvantage that they are "demand driven". Data is moved across the network, not when it is first available, but when it is first needed. The fact that a page is required is detected at page-fault time, and while the page fault handler pulls the page across the network, the faulting process is idle. Traditional DSVM systems do not provide

a convenient mechanism that lets an application initiate the transfer of data across the network in anticipation of its actual use. A sophisticated application could conceivably use a separate "scout" thread, marching ahead of the real army, to fault in pages before they are really needed, but programming such an application would not be straightforward.

Now let us compare a DSVM application with a traditional message-passing application running on the same network of machines. The message passing interface allows the user to push data across the network before it is needed at the destination processors. In a well-balanced application, processes might never have to wait for data to be transferred to their processors. DSVM processes, on the other hand, always wait while data is transferred. This fundamental difference makes DSVM systems non-competitive with explicit message passing systems from the standpoint of raw performance, even before the higher overheads of DSVM as opposed to explicit message operations are considered.

On the other hand, synchronization points in an SSVM program indicate when objects have been updated and thus indicate when data should be sent to other processors in the system. The SSVM object synchronization primitive is analogous to the message-passing send primitive, making the SSVM system supply-driven rather than demand-driven. The SSVM interface has the potential to allow the construction of well-balanced, high-performance parallel applications, while the DSVM interface does not.

Structured Shared Virtual Memory

The basic components of the Structured Shared Virtual Memory (or SSVM) interface are facilities that let the programmer:

- specify the objects the program wishes to manipulate.
- specify the subset of the application's processes that are interested in particular objects.
- specify the synchronization points in the program where updates to particular objects are complete.

For our purposes, an *object* is simply an area of storage that is part of the global shared-memory region provided by the SSVM system. (SSVM does not provide an "object-oriented" programming model such as those provided by Choices [17], Clouds [20], or Orca [4].)

SSVM supports two object types, *sequential* objects and *stride* objects. A sequential object is a contiguous sequence of bytes and typically contains an entire user data structure or a portion of a user array. A stride object can be characterized as a collection of identically-sized sequential objects, separated by fixed offsets. Stride objects typically contain slices of user arrays. Associated with each object is a list of application processes that use the object. (We assume that only one application process runs on each node of the multicomputer.) In our current implementation this list must be specified when an object is first declared. Objects are assigned object identifiers which are small integers.

Object synchronization points are specified by explicit procedure calls to the SSVM system: *ssvm_synch(obj_id)*. The *ssvm_synch* call indicates to the SSVM system that the calling process has completed an update of object *obj_id*. Logically, when the *ssvm_synch* procedure returns, the contents of the specified object have been updated on all the processors that were declared to be users of the object. Of course the implementation may delay the actual updates (see the next section), but no process may see the object's old contents once the *ssvm_synch* procedure returns.

At any time, the set of all objects in the SSVM region is called the current *template*. The current template describes how the SSVM storage region is being used by the processors. The objects in the current template cannot overlap, and they must jointly cover the entire SSVM region. The current template may be changed to reflect different phases of a computation, but template change is a relatively expensive operation. As currently implemented, a template change implies a global barrier operation; all processes in the SSVM computation must execute the template change oper-

ation before any of the processes are allowed to proceed. This synchronization is required so that the SSVM servers and the user applications all have the same notion of current template.

SSVM application processes are started independently on separate machines. (In the current implementation, application processes are invoked on multiple RS/6000 workstations using the Unix[1] *rsh* command). Each process builds a local copy of the object and template descriptors and then calls the SSVM initialization routine. As part of *ssvm_initialize*, each process connects to a global SSVM server. The global server checks the consistency of the parameters (including the object and template definitions) supplied by all the processes and then causes a logical copy of the SSVM region to appear in each SSVM process' address space. In the current implementation the SSVM region is mapped at the same virtual address in each process, but this restriction is not fundamental.

Virtual Memory Optimizations for SSVM Synchronization

The implementation of an SSVM system does not require special virtual memory capabilities from an underlying operating system, but such capabilities can be used to build more efficient implementations.

An SSVM system can be implemented on top of a pure message-passing system, with the bulk of the implementation embodied in a library that is linked with application programs. The library must be able to receive and process messages asynchronously with respect to the rest of the application. (Mach satisfies this criterion because the library can spawn background threads to receive messages asynchronously.) Such an implementation would not even require a global server, provided the individual application processes could locate each other and establish connections among themselves. The *ssvm_initialize* library routine would simply allocate a zero-filled region in the local address space to serve as the SSVM region and would then establish handlers to process messages from the other processes. The *ssvm_synch* routine would simply send the entire contents of the specified object in a message to every other process using the object. A process receiving such a message would copy the new object contents into the appropriate location(s) in the SSVM region before acknowledging the message. The *ssvm_synch* routine would not return until all the acknowledgements were received.

Such an implementation may be acceptable (or even optimal) for small objects, but it can be inefficient for objects that span multiple pages. For example, if this implementation of *ssvm_synch* were applied to a 32-megabyte object, the operation would not return until all 32 megabytes had been copied to all readers of the object. Delaying the sender is bad enough, but the receivers are likely to be delayed as well, waiting at an application-level synchronization point for the sender to "release" the object.

Using virtual memory primitives available in Mach and other systems [3], it is possible to implement *ssvm_synch* in a manner that preserves the semantics of the synchronous implementation but that does not necessarily delay the application processes while the data is actually transferred. For example, *ssvm_synch* might be implemented as follows:

1. *ssvm_synch* issues a remote procedure call (RPC) to the global server specifying the particular object to be updated.
2. The global server communicates with the other users of the object, causing them to temporarily remove all pages containing any part of the object from their address spaces.
3. The global server executes an RPC return to *ssvm_synch*.
4. *ssvm_synch* changes the protection of all pages of the object to read-only on the local machine.
5. *ssvm_synch* initiates the asynchronous transfer of the updated object to the global server.
6. *ssvm_synch* returns to the user.

[1] Unix is a registered trademark of AT&T in the United States and other countries.

7. As pages of the object are transferred to the global server, the SSVM runtime routines convert them from read-only back to read-write in the local address space.
8. As pages of the object are transferred from the global server to the destination processors, the pages are restored to the destination address spaces.

This approach preserves the semantics of the synchronous implementation of *ssvm_synch* since the destination processors cannot access the old version of the object once *ssvm_synch* has returned to its caller. Step (4) of the above procedure is required to ensure that the source process does not modify the object again before the original update has been completed. An attempt to modify (source processor) or access (destination processors) a page that has not yet been transferred results in a page fault that is caught by the library linked with the faulting process. At that point the application process is blocked until the required page is available. A sophisticated SSVM implementation might try to change the order in which pages of the object are transferred in response to such page faults. It might even, as a matter of policy, postpone transferring some pages indefinitely, or until such time as an application process requires them.

Clearly, the overhead of this asynchronous implementation of *ssvm_synch* is non-trivial and will only be justified for objects considerably larger than a page. Also, the asynchronous approach will be worthwhile only if the destination processors are usually able to continue computation on other portions of the address space during the execution of the data transfer phase of *ssvm_synch*. Therefore a different approach might be required for stride-type objects that touch large portions of the SSVM region.

In summary, we point out the advantages of the SSVM system over traditional DSVM systems:

- SSVM solves the false sharing problem since objects are not updated on other processors until an *ssvm_synch* call is executed.
- SSVM allows sharing of non-page aligned objects.
- SSVM allows efficient sharing of small objects. (The SSVM system can choose at *ssvm_synch* time to use page-oriented or message-oriented transport mechanisms to send the updates. A DSVM system must always send an entire page.)
- SSVM allows the expression of supply-driven data transfer; equivalently, the interface allows the SSVM system to perform aggressive page prefetching in response to user demands.
- SSVM allows the application of virtual memory optimizations to the problem of data transfer for large objects.

SSVM can potentially let users develop parallel programs for distributed-memory hardware using a shared-memory progamming model without sacrificing the efficiency of a message-passing programming model.

Migrating Programs into the SSVM Environment

One of the advantages of the SSVM paradigm for parallel programming over the traditional message-passing paradigm is that the shared-memory programming model of SSVM lets a programmer convert existing programs to the SSVM environment with relative ease. To justify this claim, we discuss in this section how a user can methodically convert an existing program for a shared-memory multiprocessor to run under the SSVM system. Techniques for converting existing sequential programs to run on traditional shared-memory multiprocessors are well-known [11]. For the purposes of this discussion, we assume the user begins with a correct program for a shared-memory multiprocessor (or for that matter, for a DSVM system). We further assume that the program's shared variables can be collected into a single contiguous region that can be mapped onto the shared-memory region provided by the SSVM system.

The first step in converting the multiprocessor (MP) program is conversion of the synchronization primitives. MP programs are typically coded to a particular application-level synchronization model such as semaphores, locks, or barriers. The application-level primitives are in turn implemented using hardware-specific operations such as *test-and-set* or *compare-and-swap* that atom-

ically read and modify words located in shared storage. The weak consistency model provided by SSVM is not appropriate for these synchronization variables, so the application-level synchronization primitives must be replaced with RPC versions that call the SSVM global server. Because the global server has a copy of the latest updates issued against the SSVM region, the server can execute the necessary atomic instructions against its copy of storage. On return, the application-level RPC stub procedures can copy the updated values of the synchronization variables into the local copy of the SSVM region. Assuming the original MP program never accessed synchronization variables directly, the program should continue to execute correctly after this conversion.

The second part of the conversion process involves creation of the SSVM templates and objects and the insertion of *ssvm_synch* calls. Identifying the objects in the program involves examining each call to an application level synchronization primitive and identifying the area or areas of storage the primitive protects. Each such area must be defined as an SSVM object.

Once the program objects have been defined, the insertion of *ssvm_synch* calls is straightforward: *ssvm_synch* calls are placed before each synchronization primitive that "releases" one or more objects.

In the following example, barrier #2 in process 1 releases object A, while the same barrier in process 2 releases object B.

```
Process 1                Process 2
-----------              -----------
. . .                    . . .
barrier #1;              barrier #1;
. . .                    . . .
update object A          update object B
. . .                    . . .
barrier #2;              barrier #2;
. . .                    . . .
access object B          access object A
. . .                    . . .
barrier #3;              barrier #3
. . .                    . . .
```

We therefore insert *ssvm_synch* calls before barrier #2 in both processes as follows:

```
Process 1                Process 2
-----------              -----------
. . .                    . . .
barrier #1;              barrier #1;
. . .                    . . .
update object A          update object B
. . .                    . . .
ssvm_synch(A);           ssvm_synch(B);
barrier #2;              barrier #2;
. . .                    . . .
access object B          access object A
. . .                    . . .
barrier #3;              barrier #3
. . .                    . . .
```

Now when process 2 accesses object A after passing barrier #2, it will see the correct value because process 2 cannot pass barrier #2 until process 1 reaches the same barrier, and process 1 cannot reach the barrier until its *ssvm_synch* of object A returns. When the *ssvm_synch* call returns, the updated value of object A will have been propagated (logically, at least) to process 2.

If process 2 were to access object A between barriers #1 and #2, it might or might not see the changes made by process 1, and the SSVM program would not behave correctly. But we know that process 2 cannot access object A between barriers #1 and #2 because if it did, the original MP program would not have been correct.

Note that our SSVM interface does not include any notions of locking that are directly associated with the SSVM objects (cf. the access modes associated with segments in the distributed shared memory controller operations provided in Clouds [20]). Instead we assume that synchronization is handled independently since the user began with a correct program for a shared memory MP. (However, as a performance optimization it may be useful to combine the RPC's for the *ssvm_synch* operation and the subsequent application-level synchronization primitives.)

The effort needed to convert an existing MP program to the SSVM environment will be greater than the effort required to convert the same program to a traditional DSVM environment, because the latter environment does not require the identification and definition of objects or templates. But the effort should still be considerably less than that required to convert an MP program to a message-passing environment, and the performance improvement available under the SSVM system should justify the additional effort.

Furthermore, the initial conversion of an MP program to the SSVM environment need not make sophisticated use of the SSVM facilities. The program can be converted quickly and then effort can be devoted to optimizing just those portions of the program that are performance critical. For the message-passing paradigm on the other hand, the entire program must be converted before any of it can be run. Our experience with RP3 [9] justifies this claim. We found that the initial process of migrating programs into the parallel programming environment was relatively simple. Time could then be spent restructuring those parts of the program that were crucial to obtaining good performance on the parallel machine.

We hope that parallelizing compilers such as PTRAN [2] will eventually reduce or eliminate the programming effort required to identify and define templates and objects. Alternatively, we believe that the explicit data partitioning statements of languages such as Fortran-D [15] can be used to generate the template and object information.

Approaches to Implementing SSVM under Mach

Thus far we have described the SSVM interface without discussing its implementation in any detail. Before we discuss the Mach implementation, let us consider the requirements an SSVM implementation places on the underlying operating system. SSVM software must be able to perform all of the following functions:

1. It must be able to apply asynchronous updates to user address spaces.
2. It must be able to suspend and resume the application process.
3. It must be able to manipulate page protection in the user address space.
4. It must be able to detect when the application takes a protection exception and be able to restart the user after correcting the protection exception.

Of these requirements (1) and (2) are required for any implementation of the *ssvm_synch* primitive; (3) and (4) are required for the asynchronous implementation of *ssvm_synch.*

Under Mach there are two basic approaches to supporting the SSVM system: An implementation can use the memory object (external pager) interface [22], or it can use the Mach exception mechanism [8]. Since we are in the business of supporting shared virtual memory, and since the standard shared virtual memory implementation under Mach uses the memory object interface, we expected the memory object interface would be the method of choice. We found that the choice is not so clear cut, as we will explain later in this section. In later subsections we outline implementations using both methods, but we begin by describing the common features of the two implementations.

Common features of the SSVM implementations. Both SSVM implementations use a single global synchronization and data server, and both rely on an SSVM library linked with each application process. In both cases, the global data server maintains a copy of the entire SSVM region by applying all *ssvm_synch* operations to its own copy of the SSVM region. Therefore, the global data server can always provide the latest "correct" values for a page.

The SSVM library contains application-level synchronization primitives, and the *ssvm_initialize* and *ssvm_synch* procedures. The *ssvm_initialize* routine is responsible for establishing a connection with the global server and for passing the application's template and object definitions to the server. The *ssvm_synch* routine is reponsible for collecting object data and shipping it to the global server. Sequential objects are sent directly. Stride-type objects are first copied to sequential storage and then sent. The data must be copied from its sequential form back into the SSVM region by the global server and by each recipient application task.

When the global server receives an RPC requesting an *ssvm_synch* operation, it forwards the request to each application process that is a user of the object. While the call from the application to the global server is a true RPC, the calls from the global server to the destination processes are sent as unidirectional messages so that multiple simultaneous transfers from the global server can be initiated. These messages are explicitly acknowledged by the destination processes, and the global server completes the original RPC request once it has received all the acknowledgements.

Supporting SSVM using the memory object interface. The Mach memory object interface is intended to allow the construction of memory servers that reside outside of the Mach kernel [22]. The interface is defined in [22] and [5]. Here we review the aspects of the interface that are important for this discussion.

A memory object server (or external pager) is a user mode program that provides the contents of pages for a particular memory object. A memory object is represented by a *memory object port* which is a port whose receive rights belong to the memory object server. The memory object is mapped into a task by including the memory object port in a Mach *vm_map* system call. The *vm_map* call also establishes the virtual address of the object in the task. Once the object has been mapped, a reference to a page in the *vm_map*'ed region that cannot be resolved by the kernel results in an RPC to the memory object server's *memory_object_data_request* entry point. The server does whatever is necessary to bring the data for the page into storage and then returns the data to the kernel using a *memory_object_data_provided* call.

Other routines of interest to us here are:

memory_object_lock_request
: This call from the server to the kernel allows the memory object server to reclaim pages from the kernel's page cache, to change the level of protection on objects in the cache, or to cause pages to be flushed from the cache. If a page has been modified and the server requests that it be removed from the cache, the page is delivered back to the server via a *memory_object_data_write* call.

memory_object_data_unlock
: This call from the kernel to the server occurs when an access has been made to a page that is not permitted by the current page protection. This request from the kernel asks the server to either change protection (with a *lock_request* call) to allow the requested access or to deny this mode of access.

memory_object_data_write
: This call from the kernel to the server indicates that a particular page has been modified but now needs to be removed from the kernel's page cache. Such an eviction can happen as a result of a *lock_request* call made by the server, or because the kernel's supply of free page frames has run low.

Note that in general the memory object server has no direct access to the user address space. In particular, the server has no direct way to deliver pages to the user address space before the user has touched them. The Mach 2.5 memory object interface lets the server supply more than one

page in response to a *data_request* from the kernel, but pages must be contiguous and must start at the offset requested by the kernel.[2]

Also note that the memory object interface is (understandably) page oriented. There is no simple way to update part of a page in the application address space. (If the page is already in the kernel's cache, it can be reclaimed by the server using a *lock_request* call. The page can then be updated with the new data and cached in the server address space until the user again accesses the page. There is no way to return the page to the application address space before the user next touches it.)

Another problem with the memory object interface is that it is slow, at least from the standpoint of parallel processing. Measurements on our Mach 2.5 system on the RS/6000 indicate that it requires approximately a millisecond to resolve an application page fault even when the memory object server is on the same machine as the application and when the required data is already in the server's address space. (Remember that we are trying to compete with message-passing applications that can transfer data from local message queues into a user address space in a few hundred instructions!) While an overhead of 1 ms. during the processing of a page fault that requires 30 ms. to fetch the data from a disk may not be excessive, it is very large when the page can be fetched across a fast network in 3 ms. or less.

To minimize the effect of these problems, our memory object implementation of SSVM is more complicated than we would like. The global SSVM server does not itself serve as the memory object server for the application processes. Instead we run a separate memory object server on each node of the SSVM system. Each individual server defines a memory object that serves as the SSVM region for the application task running on the same node as the server. The local memory object servers communicate with the global SSVM server using standard Mach IPC; the messages exchanged between these servers are of arbitrary size and need not be page aligned. *ssvm_synch* operations initiated on remote nodes are forwarded, not to the application processes directly, but to the local memory object servers.

The local memory object servers let the global SSVM server push new pages and page updates out to the machines running application processes, but the memory object interface by itself does not let the local servers move the data into the client address spaces before the clients need it. To circumvent this problem, each local memory object server maps the SSVM region, not only into its client's address space, but into its own address space as well. The result is that the memory object server and the application process share the storage that backs the memory object. Now when the local server gets an *ssvm_synch* request and the associated data, it can *bcopy* the data directly into the SSVM region. Of course, in doing so, it might cause page faults that result in *memory_object_data_request* calls to itself, and it must therefore take care to avoid a variety of possible race conditions and deadlocks. In particular, the server may have to suspend the application process to keep it from using out-of-date data that becomes temporarily visible. Unfortunately, the cost of handling these phantom *data_request* calls (about a millisecond per page) negates the benefits of pushing data into the application address space before it is needed.

Supporting SSVM using the Mach exception mechanism. The Mach exception mechanism [8] is a facility that allows general user-level handling of exceptional conditions. Associated with each thread in the Mach system is a special port called the thread's *exception port*. Initialized by the system to PORT_NULL, a program can designate a particular port to serve as a thread's exception port. When a thread whose exception port has been defined encounters an exceptional condition (illegal instruction, protection violation, divide by zero, etc.), the kernel suspends the thread and issues a *catch_exception_raise* RPC on the thread's exception port. (It is the program's responsibility to establish a listening thread for the exception port that will receive this RPC.) Inside *catch_exception_raise*, the program can examine the state of the suspended thread and determine whether the exceptional condition can be cleared and whether to resume or to terminate the thread.

---

2 Even this facility does not work in the Mach 2.5 kernels available to us. Our kernels hang or crash if a server tries to provide more than one page in response to a *data_request*. Joseph Barrera at CMU has indicated to us that fixes for this problem are available from CMU.

If the exception can be cleared, the handler can restart the thread that raised the exception by returning KERN_SUCCESS from *catch_exception_raise*. If the exception cannot be cleared, the handler can cause the thread to be terminated by returning KERN_FAILURE.

The Mach exception mechanism allows an implementation of SSVM that includes just one server, the SSVM global server; local servers on the application nodes are not required. In this implementation, more of the SSVM functionality is handled directly by the SSVM library linked with the application processes. The *ssvm_initialize* library routine creates the SSVM region using the standard Mach *vm_allocate* primitive. It then uses the Mach *vm_protect* call to reduce the protection on the entire SSVM region to VM_PROT_NONE (no access allowed). Finally, it establishes an exception port for the application thread and creates a new thread to handle exceptions raised by the application. Another thread is created to listen for incoming *ssvm_synch* requests from the global server.

Now when the application touches an inaccessible page in the SSVM region, it generates a protection exception which is caught by the handler thread. By examining the faulting virtual address, the handler can determine whether the fault was taken on a page in the SSVM region. If so, it calls the global SSVM server to obtain a current copy of the required page. The handler changes the page protection (again using *vm_protect*) to VM_PROT_ALL (all access allowed), copies the current data into the page, and returns KERN_SUCCESS to allow the application thread to continue.

Implementation of *ssvm_synch* is much simpler in this environment than in the memory object version. As before, *ssvm_synch* is an RPC from the application to the global server. The global server forwards the requests directly to the appropriate application tasks. Recall that *ssvm_initialize* created a thread in each application address space to listen for such requests. The global server returns to the originating *ssvm_synch* routine once it has received acknowledgements from all the helper threads. The asynchronous version of *ssvm_synch* uses *vm_protect* to make pages inaccessible in the target address spaces and read-only in the originator's address space.

Since all data is delivered directly to the application tasks (instead of to separate local servers) it can simply be copied from the RPC buffers to the SSVM regions where it belongs. Before starting the copy, the program examines the state of each target page and reprotects those that are not writeable. As in the memory object implementation, it is important to suspend the application thread before making the pages writeable. Otherwise the application may see out-of-date values before they are replaced.

Comparison of the two implementation approaches. The SSVM implementation based on the Mach exception mechanism was simpler and easier to develop than the memory object version. It has fewer components and fewer complicated interactions among components. Furthermore, the system overheads are lower in the exception port implementation for some common SSVM operations. For example, consider the time required to insert a previously absent page into an application task's SSVM region once the contents of the page have been delivered to the application's machine. In the exception port implementation, this time is roughly 700 microseconds on an RS/6000 Model 530 running Mach 2.5. The equivalent time in the memory object implementation is roughly 1200 microseconds, largely because it is impossible to avoid a *data_request*/*data_provided* trip through the memory object interface, even with the SSVM region mapped directly into the local server's address space. The difference is more dramatic for multipage transfers, because much of the exception port overhead is constant, independent of the number of pages involved.

The previous times were for inserting a page into the application's address space before the application thread needed it. If the application actually faults on an absent page, the overhead (not counting the page transfer time) of the two implementations is approximately the same (about a millisecond in each case). This overhead may be acceptable given the 11 milliseconds currently required to fetch a page from the global server. It may no longer be acceptable when the page fetch time is reduced to the anticipated 3 millisecond range.

By timing the exception path through the kernel, we have determined that for the exception port SSVM implementation, nearly half of the 1 millisecond overhead is due to suspending the application thread, invoking the handler thread, and returning back through the kernel to resume the application thread. To reduce this part of the overhead, we propose the notion of a *page fault reflector* analogous to the system-call reflector that is part of Mach 3.0. A task would register an address range and the entry point of a page-fault handler with the kernel. When the kernel detects a page fault or protection violation in the identified address range (the range would be set to the boundaries of the SSVM region) the kernel would simply resume the faulting thread but with its program counter set to the specified entry point. Information describing the fault would be pushed on the faulting thread's stack. The page fault handler would clear the fault by adjusting the protection of the faulting page and fetching current data from the global SSVM server. It would then return directly to application code without going back through the kernel. Using this approach we expect to reduce the 1 millisecond overhead mentioned above by a factor of 2.

Status and Performance Measurements

As of this writing, we have completed initial SSVM implementations using both the memory object and the exception port methods. The implementations differ slightly from those described previously in that:

1. They use a separate synchronization server rather than a server integrated with the global data server.
2. They implement only the synchronous version of *ssvm_synch*.
3. They do not yet support demand-driven page fetching. The SSVM regions are always fully instantiated.
4. The memory object implementation does not handle kernel page-out requests for the SSVM region.
5. The exception port implementation does not include the "page fault reflector".

Item (1) implies that migration of MP programs into our current environment is less transparent that it might be. Item (2) implies that for large objects we are paying an overhead at *ssvm_synch* that is higher than necessary. Item (3) will not be a problem until we have an asynchronous implementation of *ssvm_synch*. Item (4) only applies if the problem size is so large that it cannot reside in main storage, a situation we have not yet encountered. Item (5) is a performance enhancement that will become more important when our page transfer times are smaller than they are now.

Our test environment is illustrated in Figure 1. The SSVM Testbed consists of four IBM Risc System/6000 Model 530 workstations connected by IBM Risc System/6000 Optical Links to a Risc System/6000 Model 930 server in a "star" configuration. The Risc Systen/6000 Optical Link is a 200 Mb/s. optical communications link; we run TCP/IP across this link with a nominal page transfer time between machines of 11 ms. For these experiments, the Model 930 server was used solely as a host for the synchronization server and the SSVM global data server (or for the Mach Netmemory server); the Model 930 did not take part in the computations we describe below. Each of the RS/6000 workstations in the SSVM Testbed was running the X126 version of the Mach 2.5 kernel.

Computations are distributed on the Testbed using the Unix *rsh* command. One copy of the test program runs on each of the 4 Model 530's in the configuration. As part of initialization, the test programs connect to the various servers required for a particular experiment. Command line parameters are used to assign logical processor numbers to the test programs. The machines ran in multiuser mode during these test runs, but they were dedicated to these experiments; no users were running on the machines during the timing tests.

Our initial test programs are *mp_matmul*, an MP matrix multiply program, and *odd_even*, a program designed to cause page thrashing in a traditional DSVM environment.

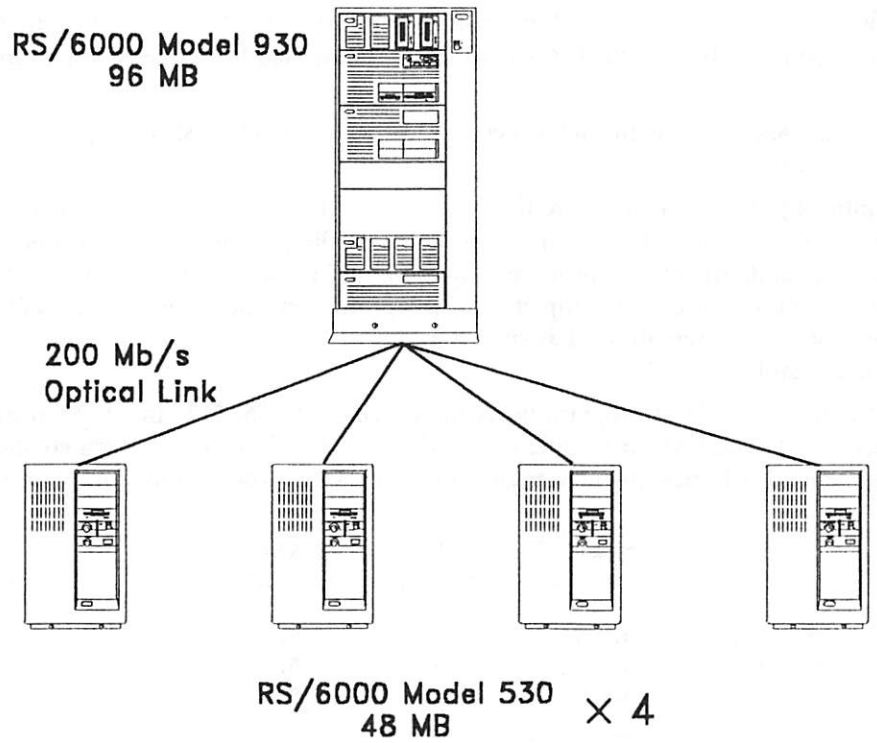


Figure 1. The SSVM Testbed

mp_matmul: This program implements a standard multiprocessor matrix multiply of M by M matrices: $C \leftarrow A \times B$. All three matrices are stored in row-major order. The computation is distributed in a straightforward manner: M/N rows of A and all of B are copied to each of the N processors (N is 4 in our case). Each processor then computes M/N rows of the product matrix C.

The computation consists of 4 major phases separated by barriers:

1. Initialization
2. Multiplication
3. Verification
4. Termination

During the initialization phase, process 1 initializes the A and B matrices. During the multiplication phase, each process uses the B matrix and its M/N rows of the A matrix to compute the corresponding M/N rows of the C matrix. During the verification phase, process 1 examines the product matrix to validate the result. When checking is complete the programs terminate.

During the computation, one slice of the A matrix moves through the Netmemory or SSVM system from process 1 to each of the other processes. The entire B matrix moves from process 1 to the other processors. At the end of the computation, one slice of the C matrix from each of the processes moves back to process 1 to be checked for correctness.

For the SSVM experiments, the matrices are partitioned into objects as follows:

The A matrix is divided into N objects, each consisting of M/N rows of the matrix.

The B matrix is also divided into N objects, each consisting of M/N columns of the matrix. These are stride-type objects because the matrix is stored in row-major order.

The *C* matrix is divided into N^2 objects, each consisting of an M/N by M/N square submatrix of *C*. These are also stride-type objects. Each *C*-object can be computed from one *A*-object and one *B*-object.

In all cases, the user-list associated with each object is defined to include just those processes that require access to the object.

This partitioning into objects is actually more fine-grained than necessary for this problem. (The *B* matrix could be a single object, and the *C* matrix could be partitioned along the same lines as the *A* matrix.) The finer granularity lets the processes initiate transfers (via *ssvm_synch*) before the initialization and computation phases are complete. This optimization provides little benefit given our current synchronous implementation of *ssvm_synch*, but it could be important when an asynchronous version is available.

Results: We ran the *mp_matmul* test program using matrix sizes of 256, 512, and 1024 in each of three environments, the standard Mach Netmemory environment and our two implementations of the SSVM environment. Each experiment was run three times, and the average times are shown below in Table 1.

Matrix size		Netmemory	SSVM mem. obj.	SSVM exc. port
256x256:	initialize	3.2 s	9.1 s	8.7 s
	multiply	14.3 s	6.3 s	6.6 s
	check result	3.7 s	0.1 s	0.1 s
	total	21.2 s	15.5 s	15.4 s
512x512:	initialize	12.3 s	34.2 s	36.5 s
	multiply	93.0 s	63.0 s	62.3 s
	check result	14.7 s	0.4 s	0.3 s
	total	120.0 s	97.6 s	99.1 s
1024x1024:	initialize	53.3 s	138.8 s	147.8 s
	multiply	590.5 s	468.0 s	466.3 s
	check result	88.0 s	1.4 s	1.2 s
	total	731.8 s	608.2 s	615.3 s

Table 1: Times for Matrix Multiply Example

The results show that the SSVM approach is 20 to 30 percent faster than the DSVM approach as embodied in the Mach Netmemory server. The reason is largely that the SSVM system batches page transfers while the Netmemory case fetches individual pages across the network. (This is an instance where the *data prefetch* advantage of the SSVM approach is clearly profitable.)

For the matrix sizes used above, false sharing does not occur since no page of the *C* matrix contains objects that are written by more than one processor. Experiments with matrices of other sizes failed to exhibit performance degradation attributable to page thrashing. In this program, false sharing can only occur at the boundaries of the objects in the *C* matrix, and the order of computation is such that it is unlikely that two processors ever try to write the same page at the same time.

Under the Netmemory server, the initialization phase does not include any data transfer, the multiplication phase includes the time needed to distribute the *A* and *B* matrices to all the processes, and the verification phase includes the time needed to fetch the result matrix *C* back to process 1. Under SSVM, the initialization phase is longer because it includes the time needed to distribute the source matrices. The multiplication phase includes the time needed to return the result matrix to process 1. The verification phase is much faster under SSVM because it involves no data transfer at all.

For comparison, we observe that the uniprocessor times for the matrix multiply results given above are 9.5 seconds, 219 seconds, and 1746 seconds, respectively. Since the computational cost increases with M^3, one would expect these times to differ by a constant factor of 8. But the smaller matrix benefits much more from the RS/6000 data cache, so the factor of 8 is not seen there. The ratio between the uniprocessor times for the 512 and 1024 dimension matrices is 8 as expected. We are not using strip mining or other cache-oriented optimization techniques. Using these uniprocessor times, the speedups obtained on 4 processors under SSVM are calculated to be 0.66, 2.23, and 2.85, respectively, for the three different matrix sizes. While this is better than the Netmemory speedups (0.45, 1.70, and 2.39, respectively) there is still room for significant improvement.

odd_even: This test program is deliberately designed to cause page-thrashing in the Netmemory server case and is intended to be a worst-case comparison of DSVM versus SSVM performance. In the *odd_even* test case, each of the N processors in the system updates a portion of a one dimensional, double-precision array. Processor 1 updates elements 0, N, 2N, etc., processor 2 updates elements 1, N + 1, 2N + 1, etc., and so forth for the other processors. The time a processor spends between updates is an adjustable parameter. When the entire array has been updated, processor 1 inspects all the entries to ensure they have the expected values. As in the *mp_matmul* test cases, the initialization, computation, and checking phases of the program are separated by barriers. An SSVM stride-type object is defined to hold the data updated by each processor.

Results: We ran this program on our 4-processor Testbed with a total of 1000 data items (250 per processor), and with delays of 0 ms. and 10 ms. between updates. With no delays between updates, the program ran in about 600 ms. under the Netmemory environment and under both implementations of the SSVM environment. Page thrashing was not apparent in the Netmemory case because each processor, after obtaining a shared page, had time to complete all of its updates before the page could be claimed by another processor.

The story is different when the processors sleep 10 ms. after updating each data element. In this case the program should run in approximately 2.5 seconds (250 items, 10 ms. per item), and indeed under the two SSVM environments the program ran in 2.7 seconds. Under the Netmemory server, however, the program required 26 seconds. This example is artificial, but it shows that the effects of false sharing can be severe in a traditional DSVM environment.

Planned Performance Improvements: We intend to pursue a number of performance optimizations. At the moment, we are running TCP/IP over the optical links in the SSVM Testbed. We know that the optical link hardware is capable of transferring pages between machines at approximately 1 ms. per page. The TCP/IP version gives us 11 ms. per page. We plan to eliminate the TCP/IP protocol by using a custom protocol directly between Mach *netmsgservers* on the machines of our Testbed. We hope to achieve a page transfer time of 3 ms. or less using this approach.

We also need to implement the asynchronous version of *ssvm_synch*. Particularly in the larger matrix multiply cases, we are paying a heavy overhead for using the synchronous version of *ssvm_synch*.

Finally, for large data copies, it would be more efficient to use the Mach *vm_copy* and *vm_deallocate* routines than to use *bcopy*. Initial measurements indicate that the break even point is around 20 pages (80 KB). Copies larger than this should use the Mach primitives instead of *bcopy*.

We hope to be able to report on the results of these improvements as well as to discuss some additional applications at the Mach Symposium in November, 1991.

Suggested Changes to the Mach Interface for SSVM Support

So far, we have attempted to achieve the best possible performance for the SSVM system using the existing Mach system interface. In this section we discuss some enhancements or changes to the Mach interface that we think would improve the performance or simplify the implementation of the SSVM system.

Directed out-of-line data. The first change we would like concerns Mach IPC and out-of-line data. Currently, out-of-line data included with a Mach message shows up at an arbitrary location in the receiver's address space. In SSVM we are constantly sending parts of large structures (a few rows of a matrix, for example) and these fragments must end up at the correct virtual addresses in the destination. At present we handle this problem by copying the data after it is received. The RS/6000 can copy a page in about 100 microseconds, but this is still a hundred microseconds of overhead we would like to avoid. The point is that we know precisely where we wish the data to appear in the destination address space, and we would like to be able to specify (direct) that the out-of-line data accompanying a Mach message be deposited at a specified address. The authorization for such directed sends might be another kind of "port right" that a destination task can give out to a trusted set of other tasks.

Asynchronous insertion of pages into memory objects. For our purposes, a major shortcoming of the Mach 2.5 memory object interface is that it does not let a memory object server asynchronously instantiate arbitrary pages of a client's mapped memory object. It provides all the facilities necessary to handle client-generated page faults, but it does not provide the facilities a server needs to preclude the necessity for such faults. We are hoping this shortcoming has been corrected in Mach 3.0.

Double mapping of anonymous memory regions. In Mach, storage that is backed by an external memory object server can be mapped at more than one location in a client's address space. The same capability is not available for storage that is simply obtained from the kernel via *vm_allocate*. The exception port implementation of SSVM would be simpler if the *vm_allocate*'d SSVM region could be mapped a second time at another location in the application address space. The access privileges available to the application via the original mapping could then differ from the privileges available to the SSVM library via the second mapping. We could obtain the desired double-mapping functionality by including local external memory object servers in the exception port implementation of SSVM, but we would rather avoid this complication.

Concluding Remarks

In this paper we presented a new shared virtual memory programming model that allows the construction of more efficient parallel programs for multicomputers than did previous shared virtual memory implementations. We defined our model (called Structured Shared Virtual Memory, or SSVM), and discussed its implementation using the Mach memory object and exception port interfaces. We found that the implementation using the exception port interface was simpler, and although performance of the two implementations was similar, we believe the exception port implementation can be more easily optimized. We then presented initial performance comparisons between SSVM and the standard implementation of distributed shared virtual memory under Mach (known as the Netmemory server [14]). The SSVM approach was seen to be 20-30 percent faster than the Netmemory approach. Finally, we suggested changes to Mach kernel interface that would make support of SSVM under Mach simpler or more efficient. Further work is necessary, both in experimenting with a wider range of applications and in enhancing the SSVM implementation.

Our current SSVM implementations run under Mach 2.5. Mach 3.0 offers us several advantages, including

- improved IPC performance.
- enhanced memory object interface that appears to support asynchronous insertion of pages into client address spaces.
- shorter path lengths in the Mach exception mechanism.
- direct kernel support for inter-machine IPC.

As we complete our port of Mach 3.0 to the RS/6000, we intend to move our SSVM environment onto that platform in order to exploit these changes.

Acknowledgements

This work would not have been possible without the assistance of Bob Fitzgerald, Dan Poff, Marie Butrico and Kati Rader, all of the IBM Research Division. Bob Fitzgerald and Dan Poff are principally responsible for the existence of a Mach port to the Risc System/6000 and Bob Fitzgerald, in particular, solved a number of problems related to running Mach on the Model 930 in our test environment. Maria Butrico and Kati Rader provided us with the driver code from AIX/v3 for the Risc System/6000 Two Port Optical Link adapter, and patiently assisted us in the port of this code from AIX/v3 to Mach.

References

[1] M. Accetta et al., "Mach: A New Kernel Foundation for Unix Development," *Usenix Association Proceedings*, Summer 1986.

[2] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing," *The Journal of Parallel and Distributed Computing*, vol. 5, no. 5, pp. 617-640, Oct 1988.

[3] A. Appel and K. Li, "Virtual Memory Primitives for User Programs," *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 96-107, Baltimore, Md. Order Number 556910: ACM, April 1991.

[4] H. Bal, M. Kaashoek, and A. Tannenbaum, "A Distributed Implementation of the Shared Data-Object Model," *Proceedings of the Distributed and Multiprocessor Systems Workshop*, pp. 1-19, Usenix Assocation, 1990.

[5] R. Baron, D. Black, W. Bolosky, J. Chew, R. Draves, D. Golub, R. Rashid, A. Tevanian, Jr., and M. Young, "Mach Kernel Interface Manual", Pittsburgh: School of Computer Science, Carnegie Mellon University, Unpublished manuscript, August 1990.

[6] J. Bennett, J. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proceedings 1990 Conference Principles and Practice of Parallel Programming*, pp. 168-176, New York, N. Y.: ACM Press, 1990.

[7] R. Bisiani and M. Ravishankar, "Plus: A Distributed Shared Memory System," *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 115-124, Los Alamitos, Cal. Order No. 2047: IEEE CS Press, 1990.

[8] D. Black, D. Golub, R. Rashid, A. Tevanian, and M. Young, "The Mach Exception Handling Facility," *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, SIGPLAN Notices*, vol. 24, no. 1, pp. 45-56, Department of Computer Science, Rice University, P. O. Box 1982, Houston, Texas, May 1988.

[9] R. M. Bryant, H.-Y. Chang, and B. S. Rosenburg, "Operating System Support for Parallel Programming on RP3," *IBM Journal of Research and Development*, To appear: November 1991.

[10] A. Cox and R. Fowler, "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with Platinum," *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 32-44, December 1989.

[11] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A single-program-multiple-data computational model for EPEX/Fortran," *Parallel Computing*, no. 7, pp. 11-24, 1988.

[12] M. Dubois and C. Scheurich, "Memory access dependencies in shared-memory multiprocessors.," *IEEE Transactions on Software Engineering*, vol. 16, no. 6, pp. 660-673, June 1990.

[13] B. Fleisch and G. Popek, "Mirage: A Coherent Distributed Shared Memory Design," *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 211-223, December 1989.

[14] A. Forin, J. Barrera, M. Young, and R. Rashid, "The Shared Memory Server," *Proceedings of the 1989 Winter Usenix Conference*, pp. 229-243, 1989.

[15] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu, "Fortran D Language Specification", Houston, Texas: Department of Computer Science, Rice University, P. O. Box 1982, Technical Report COMP TR90-141, December 1990.

[16] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennesy, "Memory consistency and event ordering in scalable shared memory multiprocessors," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 15-26, May 1990.

[17] G. Johnston and R. Campbell, "An Object-Oriented Implementation of Distributed Virtual Memory," *Proceedings of the Distributed and Multiprocessor Systems Workshop*, pp. 39-57, Usenix Assocation, 1990.

[18] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Transactions on Computer Systems*, vol. 7, no. 4, pp. 321-359, November 1989.

[19] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer*, vol. 24, no. 8, pp. 52-60, August 1991.

[20] U. Ramachandran and M. Khalidi, "An Implementation of Distributed Shared Memory," *Proceedings of the Distributed and Multiprocessor Systems Workshop*, pp. 21-38, Usenix Assocation, 1990.

[21] M. Stumm and S. Zhou, "Algorithms Implementing Distributed Shared Memory," *IEEE Computer*, vol. 23, no. 5, pp. 54-64, May 1990.

[22] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, B. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proceedings of the 11th Symposium on Operating Systems Principles*, pp. 63-76, Austin, Texas: ACM, November 1987.

Managing Discardable Pages with an External Pager

Indira Subramanian
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Internet: indira@cs.cmu.edu

Abstract

We have designed and implemented an *external pager* that: (1) receives information regarding discardability of pages from the client such as a garbage collector for functional programming languages, (2) saves and restores only non-discardable pages and (3) influences page-replacement by pre-flushing discardable pages. In a general purpose operating system dirty pages are typically saved to disk and then restored from disk even when the application may not care about the contents of those pages. For example, copying garbage collection used in functional programming languages generates many pages that are dirty but discardable. Using the external pager to manage the discardable pages we observed that elapsed times for some applications decreased by as much as a factor of 6.

1. Introduction

Saving and restoring discardable dirty pages causes unnecessary disk traffic, thus preventing computer systems from achieving the best possible performance. An application may modify one or more pages during the course of its execution. At subsequent points of execution, the application may not care about the contents of some of these modified (dirty) pages. We refer to these modified pages as *discardable dirty* pages. The ratio between the average access time of magnetic disk and the average access time of main memory is often between 2,500 and 70,000 [Gibson 90]. Hence, saving and restoring discardable dirty pages would undermine the throughput of a computer system.

One of the applications that can take advantage of discardable page management is copying garbage collection. In systems that use copying garbage collection, such as the one described in [Appel 89], heap space is partioned into a *new*, *old* and *reserved* regions. Allocations are made out of the new space. When the new space is exhausted, a *minor* collection phase is initiated. During the minor collection, *live* data are copied from the new space to the reserved space, which immediately follows the old data. The old region continues to grow into the reserved region. Once the old region reaches a certain size, a *major* collection phase is initiated. During the major phase, the live data from the old region is copied to another region (in the heap), which becomes the old region from then on. At the end of each minor and major collections, several pages become discardable. Examples of systems that use copying garbage collection are: SML/NJ garbage collector [Appel and MacQueen 87], and the CMU Common LISP garbage collector [MacLachlan 91].

Another application domain that can benefit from support for discardable pages is image processing (e.g., image restoration, image compression, and texture synthesis). Texture synthesis is an important component of computer graphics as well as image compression. One of the techniques that has been developed for texture synthesis is a two stage procedure [Balram 91] involving matrix computations. This technique generates several intermediate matrices that become discardable. For example, consider the case

---

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDS972-90-C-0035. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, or the U.S. government.

where a typical image represented as a 512x512 double precision matrix is to be synthesised. At the end of the first stage, there will be 26 512x512 matrices of double precision data, of which 20 matrices are input to the second stage and 6 matrices become discardable. Hence, in the first stage, the total memory requirement is 52 megabytes of which 12 megabytes are discardable. In the second stage, there will be 21 matrices, of which 20 are from the first stage and 1 matrix is the output matrix of the second stage. As the iterations in the second stage proceed, each of the 20 input matrices become discardable. Support for managing discardable pages is important since images can be even larger. The ability to discard pages would also be useful in applications such as smoothing of noisy images, where a similar two stage procedure is used.

In Mach, the vm_allocate and vm_deallocate calls can be used to map or unmap pages in a task's address space [Rashid et. al. 88]. The vm_deallocate call could be used to unmap the pages that an application would like to discard. However, the physical pages get freed only if an entire object allocated by vm_allocate is being deallocated (by the vm_deallocate). If only a part of an object is deallocated, the pages are unmapped from the client's address space, but they remain as part of the object. Hence, vm_deallocate does not discard the pages unless a client deallocates an entire object. In fact, the copying garbage collector in CMU Common Lisp uses the vm_allocate call to allocate the new region; at the end of minor collection, this new region is deallocated by vm_deallocate call. Even modifying the kernel to solve this problem introduces additional difficulties because repeated vm_deallocates and vm_allocates can result in the fragmentation of a memory object, degrading memory management performance.

To the best of our knowledge, general purpose operating systems do not provide support for identifying discardable dirty pages as such and for avoiding the unnecessary saving and restoring of those pages.

2. Solution

Our solution is to exploit the knowledge about discardable pages in an application's address space and avoid unnecessary saving and restoring of those pages. Many applications, such as garbage collectors and image processing systems know what pages are discardable at any point in time. These applications can communicate this information to the operating system.

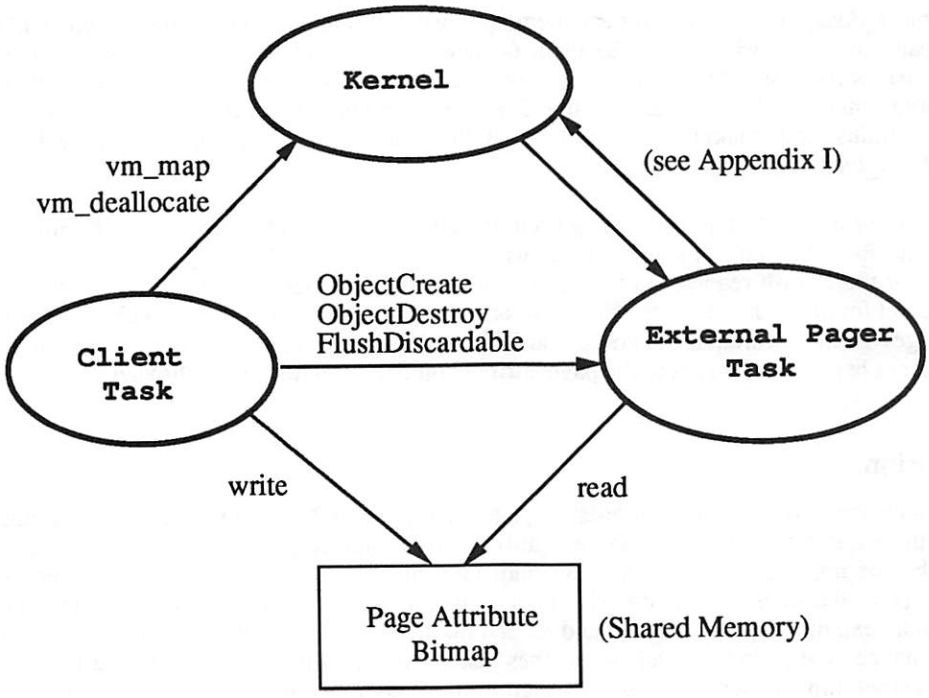
We have implemented under the Mach Operating System [Tevanian 87a], a pager that facilitates efficient management of discardable pages. Mach allows the user to create *memory objects* [Tevanian 87b] that can be managed by a user-level process, called the *external pager* [Young et. al. 87]. Through the external pager, a client may create an object and request that the kernel map the object into its address space. Figure 2-1 summarizes the interaction between the kernel, the pager and a client.

2.1. The Page Attribute Bitmap

The information regarding the discardability of pages in the memory object created by *ObjectCreate* is maintained in a bitmap shared between the client and the pager. This state information is written by the client and read by the pager; it contains one bit per page, indicating that the page is in one of two states:

- non-discardable - when paged out by the kernel, the pager must save the contents of this page to backing storage; when paging in, the pager must retrieve the contents of the page from the backing storage
- discardable - when paged out by the kernel, the pager may discard this page; when paging in, the contents of the page are irrelevant to the client

The page attribute bitmap, also called the *attribute object*, is itself a memory object created by the pager. The memory object whose pages will be managed by the pager, is called the *primary object*. A call to *ObjectCreate* returns two ports, one for the primary object and the other for the attribute object. The client then maps the primary object into its address space by calling *vm_map*, which results in the kernel calling *memory_object_init*. In *memory_object_init*, the pager maps the attribute object into pager's address space. Upon returning from *vm_map*, the client calls *vm_map* again to map the attribute object into client's address space. In this manner, the attribute object is shared between the client and the pager. The primary object and the attribute object are managed by two different threads running within the pager to avoid



Client to Kernel

vm_map	maps a memory object into client's address space
vm_deallocate	unmaps a specified range of pages from clients address space

Client to Pager

ObjectCreate	creates a memory object
ObjectDestroy	terminates a memory object
FlushDiscardable	flushes all discardable pages in a memory object

Figure 2-1: Interactions: Messages and Shared Memory

deadlock.

2.2. Improving Paging Performance: Pre-flushing Strategy

The pager achieves significant performance gains in two ways. First, by using the information in the shared bitmap, it can avoid saving and restoring discardable pages. Second, it can pre-flush discardable pages, allowing non-discardable pages to remain in memory for longer periods. This further improves performance because it is much less expensive to flush several discardable pages than to write a non-discardable page to backing store, and to read that page back in later. By default, the Mach kernel uses a global page replacement policy based on a modified First-In-First-Out strategy, which does not take into consideration whether a dirty page is discardable or not. We use two approaches for pre-flushing: *user-initiated* and *pager-initiated.*

In the user-initiated pre-flushing technique the client task, sets up the page attribute bitmap and explicitly requests the pager to flush discardable pages by calling FlushDiscardable. The pager then calls *memory_object_lock_request* to flush all the pages that are marked discardable.

In the pager-initiated pre-flushing technique, the pager calls *memory_object_lock_request* to flush a predefined number of discardable pages when it receives a *memory_object_data_write* call from the kernel. This is an appropriate time to flush discardable pages because if the kernel is making

memory_object_data_write calls, physical memory must be tight. In the current implementation, each time, the pager makes flush requests to flush 64 pages. We chose 64 because pre-flushing too many discardable pages could add to the execution time; pre-flushing too few pages would mean that the kernel could pageout non-discardable pages. At this time, we have not experimented with values other than 64. The pager refrains from making flush requests if there are acknowledgements pending from previous *memory_object_lock_request* calls.

When the pre-flushing technique is being used, the client and the pager need to synchronize their access to the page attribute bitmap. A single lock is used for this purpose. The pager acquires the lock before making one or more flush requests to the kernel and releases the lock after it receives acknowledgements from the kernel for all of the requests. The client acquires the lock when it needs to change the state of one or more pages from discardable to non-discardable and then releases it. Note that when pre-flushing technique is not being used, access to the page attribute bitmap need not be synchronized.

3. Evaluation

Performance measurements of discardable page management by an external pager is important for answering three questions: First, are there significant performance gains to be realised even if only in a small number of important applications? Second, how many applications can benefit from support for discardable page management? Third, should the support for discardable page management be in the kernel, or in an external pager? The first and second questions will be answered as we build our experience with performance of applications such as the ones discussed in section 1. Even if the performance benefits from an in-kernel implementation are greater than from an external pager, we need to be convinced that (a) an in-kernel implementation is simple (b) there are enough applications to warrant an in-kernel implementation. In this section, we will present some data on the performance measurements we made in the context of a copying garbage collector.

3.1. Experimental Setup

The measurements that are reported here were made on a Digital Equipment Corporation's Decstation 5000/200 with 24 megabytes of physical memory, running Mach version 2.5. This is a typical configuration in our environment. For the purpose of running the tests, the machine was booted in single user mode. The only other programs running were a nameserver and the external pager. The nameserver is used only at client startup, to look up the external pager service port. Hence for all practical purposes, only the client and the external pager were running in the system.

We ran four configurations: (a) without pager, (b) with pager, not using pre-flushing, (c) with pager, using pager-initiated pre-flushing, and (d) with pager, using user-initiated pre-flushing. They are summarized in table 3-1. In each test case, each of the four configurations were run 6 times. The average measurements are being reported in the performance tables. The entries in the performance tables are described in table 3-2.

NoPager	run without the external pager
PagerNF	run with the pager, without pre-flushing
PagerPF	run with the pager, using pager-initiated pre-flushing only
PagerUF	run with the pager, using user-initiated pre-flushing only

Table 3-1: Configurations

3.2. Performance Measurements

Two test cases written in Standard ML [Milner et.al. 90] were studied: (a) compiling parts of the SML/NJ compiler and (b) a sort program. Modifications were made to the ML garbage collector to use our external pager to create and manage the heap memory.

Time	shown in minutes:seconds
Pageins:Read	number of pages read from disk
Pageins:NoRead	number of discarded pages returned to client as zero-filled
Pageouts:Write	number of non-discardable pages written to disk
Pageous:Discard	number of discardable pages written to pager
Pageouts:Flush	number of discardable pages requested of kernel to be flushed

Table 3-2: Performance Metrics

Besides exploiting support for discardable page management, the ML garbage collector does some memory management of its own. It takes a set of parameters called *gc* parameters, that help maintain the working set pertaining to the heap within the available physical memory [Cooper and Nettles 91]. The gc parameters have effect on how frequently minor and major collections happen. On a 24 megabyte machine, different combinations of gc parameters could result in variations in the elapsed time of a given application. Details regarding the impacts of gc parameters are beyond the scope of this paper. Cooper and Nettles report significant performance improvements running ML applications with various gc parameters when using our pager [Cooper and Nettles 91].

Performance measurements for the compiler are shown in table 3-3. We observe that:

1. The elapsed time when not using the pager is significantly larger than the user and system times (i.e., the CPU utilization is low). As we introduce the pager and then the pre-flushing techniques, the CPU utilization increases, due to reduced disk accesses.
2. When using the pager without pre-flushing, the number of reads and writes is higher than with user or pager initiated pre-flushing. This is because in both the pre-flush cases, the kernel is able to retain more non-discardable pages.

	Time			Pageins		Pageouts		
Configuration	User	System	Elapsed	Read	NoRead	Write	Discard	Flush
NoPager	2:49	0:16	16:00	22792	N/A	21693	N/A	N/A
PagerNF	2:51	0:17	8:01	6594	15516	6525	18813	N/A
PagerPF	2:45	0:18	3:56	710	24755	710	425	27899
PagerUF	2:49	0:22	5:18	2861	33051	2861	1720	36233

Table 3-3: Measurements: Compiling Parts of SML/NJ Compiler

The elapsed time for pager-initiated flushing is smaller than for user-initiated flushing. This is because the pager-initiated flush happens only when memory gets tight, and a small number of discardable pages are pre-flushed at that time. The user-initiated flush, on the other hand, happens whenever the user chooses to activate it, and the user may have marked many pages discardable. This implies that pager-initiated pre-flush is more likely to track closely the memory demands in the system than the user-initiated pre-flush. However, our experience has shown that this is not always the case. We do not in general, have an accurate knowledge of the optimal number of pages to pre-flush from outside the kernel. This is one of the disadvantages of doing discardable page management through an external pager. In the case of an in-kernel implementation, the kernel flushes a discardable page only when it is in need of more memory. Unlike the external pager implementation, there can never be too few or too many flushes.

Another test case was a sort program. It does quicksort using ML futures [Cooper and Morrisett 90], a variant of MultiLisp futures which is a construct for creating tasks and synchronizing among them [Halstead 85]. The results are summarized in table 3-4. Once again we find that using the pager, particularly with pre-flushing techniques, reduces the elapsed time significantly. Note that the elapsed times for pager-initiated flush and user-initiated flush are almost the same in this case.

	Time			Pageins		Pageouts		
Configuration	User	System	Elapsed	Read	NoRead	Write	Discard	Flush
NoPager	1:36	0:16	24:04	27669	N/A	27524	N/A	N/A
PagerNF	1:36	0:19	8:42	10969	13439	12632	16038	N/A
PagerPF	1:37	0:16	4:00	3981	18463	5124	327	21454
PagerUF	1:36	0:18	3:58	3707	24491	5149	620	28327

Table 3-4: Measurements: Sorting using futures in ML

4. Related Work

Another example of an external pager is the PREMO pager [McNamee and Armstrong 90]. This work extends the external pager interface to implement user-level paging policies. The kernel is modified to supply the PREMO pager information on the state (such as modified, referenced etc.) of physical pages and accept from the pager the order in which the pages should be replaced.

5. Future Work

Managing discardable pages with an external pager has been shown to yield good results, especially when using pre-flushing techniques. However, it is evident that having the information about discardable pages within the kernel would have certain advantages:

- Performing too many pre-flushes is likely to impede system performance; in the case of in-kernel implementation, knowledge of physical memory availability would enable the kernel to flush a discardable page only when necessary, thereby avoiding unnecessary zero-fills
- It is possible to reduce the the number of zero-fills without compromising security. For example, in Mach, we could let the kernel keep track of which task had last used a discarded page in the inactive queue. When a task pagefaults on a discarded page, the kernel could return the discarded page in the inactive queue, which had previously belonged to the task thus avoiding the need to zero-fill the page. It is our belief that systems such as SML/NJ which page heavily against their own pages and generate many discardable pages will benefit from this approach.

To evaluate this approach, we have begun an in-kernel implementation of discardable page management under Mach version 3.0.

6. Conclusion

Given the wide disparity in the speed of primary memory and backing storage, it is important to manage primary memory efficiently. Physical memory is getting cheaper, but at the same time, we are seeing a growing number of applications that are memory intensive. In this paper we have demonstrated the need to manage discardable pages for a certain number of applications. We are in the process of identifying other applications that can benefit from support for discardable page management. Results from this effort and an in-kernel implementation will be reported in a future paper [Subramanian 91].

7. Acknowledgements

I would like to thank Scott Nettles for getting the SML/NJ working with the pager. I would like to thank Eric Cooper and Rick Rashid for suggesting the problem of discardable pages and for continued support in pursuing further work in this area. I am grateful to Jeannette Wing and Bob Baron for allowing me to use their workstations on which all the performance measurements were made.

Appendix I Kernel - External Pager Interface

Our external pager is functional under Mach versions 2.5 as well as 3.0. At the time of writing, our pager uses only the Mach 2.5 Kernel - External Pager Interface.

kernel to pager:

memory_object_init
: initialise a memory object that is being mapped to a task's address space

memory_object_terminate
: indicates that the specified memory object is no longer mapped

memory_object_data_request
: request data from this memory object

memory_object_copy
: indicates that a copy has been made of the specified range of the given original memory object

memory_object_data_unlock
: request that the specified portion of the memory object be allowed the specified forms of access

memory_object_data_write
: write back modifications made to this portion of the memory object while in memory

memory_object_lock_completed
: indicate that a previous memory_object_lock_request has been completed

pager to kernel:

memory_object_set_attributes
: make decisions regarding the use of the specified memory object

memory_object_get_attributes
: retrieve attributes currently associated with an object

memory_object_data_provided
: provide data contents of a given memory object

memory_object_data_unavailable
: indicate that zero-fill page should be provided

memory_object_lock_request
: indicate that the protection on a given range of pages be changed; may require that the data be written back to the manager

memory_object_data_error
: indicate that a range of specified memory object cannot be provided at this time

memory_object_destroy
: indicate that the pager will no longer supply data for this object

References

[Appel 89] Andrew W. Appel.
Simple Generational Garbage Collection and Fast Allocation.
Software-Practice and Experience , February, 1989.

[Appel and MacQueen 87]
Andrew W. Appel and David B. MacQueen.
A Standard ML Compiler.
Functional Programming Languages and Computer Architecture.
Springer-Verlag, 1987, pages 301--324.
Volume 274 of Lecture Notes in Computer Science.

[Balram 91] Nikhil Balram and Jose M. F. Moura.
Recursive Enhancement of Noncausal Images.
In *Proceedings IEEE ICASSP'91*, pages 2997-3000. May, 1991.

[Cooper and Morrisett 90]
Eric C. Cooper and J. Gregory Morrisett.
Adding Threads to Standard ML.
Technical Report, School of Computer Science, Carnegie Mellon University, December, 1990.

[Cooper and Nettles 91]
Eric Cooper, Scott Nettles.
Experience With Mach External Pager for ML Garbage Collection.
In Preparation , 1991.

[Gibson 90] Garth A. Gibson.
Redundant Disk Arrays: Reliable, Parallel Secondary Storage.
PhD thesis, Computer Science Division, University of California, Berkeley, December, 1990.

[Halstead 85] Robert H. Halstead Jr.
Multilisp: A Language for Concurrent Symbolic Computation.
ACM Transactions on Programming Languages and Systems , October, 1985.

[MacLachlan 91] Robert A. MacLachlan.
CMU Common Lisp User's Manual
School of Computer Science, Carnegie Mellon University, 1991.

[McNamee and Armstrong 90]
Dylan McNamee and Katherine Armstrong.
Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies.
In *USENIX Mach Workshop*. USENIX Association, October, 1990.

[Milner et.al. 90] Robin Milner, Mads Tofte, and Robert Harper.
The Definition of Standard ML.
MIT Press, 1990.

[Rashid et. al. 88] Rashid, Richard F. et. al.
Machine Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.
IEEE Transactions on Computers , August, 1988.

[Subramanian 91] Indira Subramanian.
Managing Discardable Pages.
In Preparation , 1991.

[Tevanian 87a] Avadis Tevanian Jr. and Richard F. Rashid.
MACH: A Basis for Future UNIX Development.
Technical Report, Computer Science Department, Carnegie Mellon University, June, 1987.

[Tevanian 87b] Avadis Tevanian Jr.
Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach.
PhD thesis, Computer Science Department, Carnegie Mellon University, December, 1987.

[Young et. al. 87] Michael Young et. al.
The Duality of Memory and Communication in the Implementation of a Multiprocessor Oprating System.
Technical Report, Department of Computer Science, Carnegie Mellon University, August, 1987.

OSF/1 Virtual Memory Improvements

David Black, Jeff Carter, George Feinberg,
Rod MacDonald, Shashi Mangalat, Eric Shienbrood,
Jim Van Sciver, Ping Wang

October, 1991

Abstract

The second release of the Open Software Foundation's operating system, OSF/1, is nearing completion. Among the features in this release are an increased focus on performance and robustness. This paper details the improvements made in these areas to the virtual memory subsystem. The improvements and the reasoning behind four selected changes are presented: deadlock removal in the VM code, eager allocation of backing store for kernel stacks, the addition of clustered page operations, and the addition of swapping.

1. Introduction

The first release of the Open Software Foundation's operating system offering, OSF/1, was made generally available in December of 1990. Since that initial release the OSF operating system group has been continuing its efforts to enhance the commercial viability of OSF/1 for the benefit of our members. Among the goals for this second release, termed OSF/1 R1.1, were performance and robustness improvements.

When considering subsystems for performance improvement, we wanted to choose only those areas which were deemed both critical and architecture neutral. The OSF/1 virtual memory subsystem met both of these qualifying criteria and, as a result, was the primary focus of the performance/robustness efforts in the R1.1 project.

OSF/1 is derived from Mach 2.5 technology. The Mach virtual memory subsystem has a number of desirable characteristics. Its architecture is clean and easily understood. The boundary between machine independent and machine dependent code is clearly demarcated with a well defined set of interfaces. Mach also exports an interface that allows user space applications to participate in memory management. Such an external memory manager (EMM) is responsible for managing secondary storage, supplying page data to the kernel on request, and accepting page write requests. External memory managers run in separate tasks, and communicate with the kernel using Mach IPC.

We examined the virtual memory subsystem in two ways. First, we stressed the system under increasing load and compared the OSF/1 system performance with another commercial version of Unix on the same hardware. Second, we ran small programs to stimulate specific virtual memory metrics, such as paging rates, and compared our results to the other system. OSF/1 R1.0 performed worse than the commercially available system and, in the process of load testing, a number of undesirable system failures were discovered.

This paper describes the modifications made to the virtual memory subsystem to correct these problems. The robustness changes were the removal of deadlocks when wiring pages and the preallocation of backing store for kernel stacks. Also, clustered paging was added to improve performance. The addition of swapping improves both system performance and robustness.

2. Deadlock Removal

The routine vm_map_pageable() is used throughout the kernel for wiring and unwiring memory. Three problems were found with the R1.0 version of this routine:

- it holds the map lock for the duration of the faults that wire down pages. This means that vm_fault is called with the map locked for read. Under certain conditions deadlock will result when the kernel tries to acquire the map lock for write.
- it ignores error codes from vm_fault. The system assumes that wiring down memory cannot fail. Unusual error conditions occur when it does.
- vm_map entry deletion does not synchronize correctly for wired entries. A wired entry may be unwired and deleted regardless of the wired state.

It became obvious that in order to avoid holding the map lock we needed to keep the current wired state of a map entry. A new flag, *in_transition*, was added to the vm_map_entry structure to indicate if an entry is being wired or unwired. During a wiring operation a thread will block if it encounters an in_transition entry. The blocked thread indicates that it wants the entry by setting the new flag *needs_wakeup*. The thread which set the in_transition bit will wakeup threads waiting on the entry once it completes the wiring.

If the vm_fault fails during a wiring operation all entries wired thus far are unwired and a failure status is returned to the caller. Failure to unwire an entry is a panic condition.

When unwiring an entry, either directly or by deleting an address range, wired entries are not unwired unless all the wirings are removed. The *wired_count* member is a counter indicating how many kernel wirings have been performed on the entry. An entry deletion always removes one kernel wiring and, if the wired count falls to zero, continues to unwire and deallocate the entry.

An additional wired reference count, *user_wired_count*, tracks user wirings. This count allows users to wire down memory. No comparable user interface currently exists yet to take advantage of the interface. In order to protect the kernel from loss of available physical memory, deleting user wired memory always unwires the memory, disregarding the number of user wirings. This policy guarantees that all user wired memory is unwired and deallocated when a user program exits.

The vm_map_wire and vm_map_unwire routines now take an additional argument to distinguish between kernel or user requests. A new macro vm_map_pageable_user() is provided to facilitate this. Currently only the plock system call uses vm_map_pageable_user.

By reducing the scope of the map lock during wiring/unwiring operation we achieve better parallelism and performance. The number of simultaneous wirings on a single map no longer depends on the map lock. Operation granularity is finer since synchronization needs to be done only for overlapping regions at the entry level rather than at the map level. On a multiprocessor this allows parallel wirings of the same map at different address ranges.

3. Eager Allocation of Backing Store for Kernel Stacks

In OSF/1 R1.0 and Mach 2.5 backing store allocation for a virtual page is deferred until the page needs to be paged out. This is a good performance strategy since many pages are created and destroyed without ever being paged out. However, in the case of kernel thread stacks, this policy can lead to disaster if a stack page must be paged out when there is no space available in the paging file. The pageout operation will fail but the operating system does not discover this failure until it makes a subsequent attempt to page in the kernel stack. The end result is an unrecoverable page fault when the kernel attempts to touch that

kernel stack, causing a system crash.

The solution is to allocate backing store for kernel stack space at the time the stack is created. Then either the thread creation fails gracefully because there was no more space in the paging file, or the kernel can be assured that all page faults on the thread's kernel stack will succeed.

Mach uses a client server model to perform paging instead of embedding paging in the kernel's virtual memory system. The three components of this model are the kernel's VM subsystem (client), a pager which can be either internal or external to the kernel (server), and a communications mechanism. Implementing backing store preallocation required changes to each of these three components. Allocate and free interfaces had to be added to both the client (kernel) and server (pager), and the kernel IPC mechanism needed extensions to support kernel initiated RPCs. Mach documentation usually refers to pagers as memory managers.

3.1 External Interfaces

There are two components to the interface between Mach and the pager that handles kernel created memory (e.g., kernel stacks). This pager is known as the *default pager*; this is the vnode pager in OSF/1. In addition to the standard interface used by all external pagers, this pager implements an extension to support paging of kernel created memory. The routines in this extension allow the default pager to accept responsibility for a memory object created by the kernel, and to correctly implement the paging of memory copied from another memory object that may not be managed by the default pager. The actual paging operations for kernel created memory use routines from both the standard interface and this extension.

Our implementation of eager allocation added two routines to this extension of the external memory management interface. These routines are only used in the paging of kernel created memory, such as kernel stacks.

```
kern_return_t
memory_object_data_allocate(mem_obj, mem_obj_control, offset, length)
        memory_object_t         mem_obj;
        memory_object_control_t mem_obj_control;
        vm_offset_t             offset;
        vm_size_t               length;

kern_return_t
memory_object_data_terminate(mem_obj, mem_obj_control, offset, length)
        memory_object_t         mem_obj;
        memory_object_control_t mem_obj_control;
        vm_offset_t             offset;
        vm_size_t               length;
```

The *memory_object_data_allocate()* interface asks the pager to allocate *length* bytes of backing store starting at *offset* in the specified memory object *mem_obj*. (The *mem_obj_control* argument allows the pager to identify which kernel made the request if more than one kernel is using the memory object.) Conversely, *memory_object_data_terminate()* asks the pager to deallocate previously allocated backing store. Unlike all other pager interfaces, memory_object_data_allocate is synchronous. This ensures that the backing store is allocated before the operation requiring it completes. This routine returns KERN_SUCCESS upon successful completion or KERN_RESOURCE_SHORTAGE if there is insufficient backing store. This allows thread creation to fail (synchronously) when backing store is not available for the new thread's kernel stack. The companion routine, memory_object_data_terminate(), is asynchronous. Attempts to deallocate non-existent backing store are ignored.

3.2 Internal Interfaces

Two similar routines have been added to the internal VM interfaces. These routines provide the kernel (client) side interface to backing store allocation/deallocation functions. These interfaces can be used for the allocation/deallocation of backing store for any subset of virtual space.

```
kern_return_t
vm_map_backing_alloc(map, vaddr, size)
        vm_map_t        map;
        vm_offset_t     vaddr;
        vm_size_t       size;

void
vm_map_backing_free(map, vaddr, size)
        vm_map_t        map;
        vm_offset_t     vaddr;
        vm_size_t       size;
```

The *vm_map_backing_alloc()* routine sends a request to the default pager to allocate backing store for the subset of *map* that starts at *vaddr* and has size *size*. If the backing store is successfully allocated or had been previously allocated, KERN_SUCCESS is returned. Otherwise KERN_RESOURCE_SHORTAGE is returned.

Conversely, *vm_map_backing_free()* sends a request to the default pager to deallocate backing store. This routine does not wait for a reply because the corresponding memory_object_data_terminate request is asynchronous and it is impossible for the free request to return an error. If the backing store had been previously allocated then the vm_map_backing_free() request would succeed. If the backing store is unallocated at the time of the request then the request is simply ignored.

3.3 Mach Kernel RPC

The implementation of backing store allocation required modifying the Mach IPC system to support kernel initiated RPCs. Mach IPC normally assumes that all messages sent to the kernel are operation invocations, and replaces the receive processing of these messages with a direct invocation of the corresponding operations. This makes it impossible for threads to wait in the kernel for messages. Instead, such threads synchronize with the invoked operations. A special trick was used for the exception facility; the kernel would pretend that it was a user task while sending an exception RPC. This was the only circumstance in which the kernel initiated an RPC.

Neither of these techniques is suitable for returning the result of backing storage allocation. Implementing an operation to return the result requires a separate data structure and logic that serve no other purpose. Borrowing the identity of the user task that is creating the thread (and its kernel stack) is wrong in principle (the kernel should be allocating backing store for kernel stacks), and fails in practice for the creation of kernel threads. Instead we decided to extend Mach IPC to support kernel initiated RPCs (kernel RPC).

The implementation of kernel RPC marks ports to detect reply messages to kernel RPCs. When a thread in the kernel initiates an RPC, the reply port's *kernel_reply_port* field is set to mark it as a reply port for a kernel RPC. The IPC system delivers messages to such ports normally (e.g., completing the RPC) instead of attempting an operation invocation. This change only affects kernel initiated RPCs; RPCs directed at the kernel still invoke kernel operations as before. The reply port for a kernel RPC in progress is recorded in the data structure of the thread that is waiting for the reply; this allows the RPC to be aborted if

necessary. The kernel caches and reuses these reply ports when possible. The completion of an RPC frees the reply port to a cache which is checked when a new RPC is initiated. Reuse of a port from this cache requires passing a security check to ensure that no user task has surreptitiously queued a message on or retained a send right to the port; the use of send once rights in Mach 3.0 IPC will obviate the need for this check.

We also used kernel RPC to greatly simplify and improve the implementation of Mach's exception facility. The impersonation of a user task by the kernel causes the exception facility to contain a significant amount of code that understands IPC internals. For example, the exception facility must explicitly translate port names from the kernel to the impersonated task. Using kernel RPC in the exception facility allowed all of the offending code to be removed, and simplified the exception facility; now it need only find the appropriate exception port and initiate the RPC. Using kernel RPC also removed the side effect of the exception facility allocating its reply ports in the user task's port space.

3.4 Kernel and Vnode Pager Changes

Changes were required beyond adding and using the interfaces to allocate and deallocate backing store. The first was to handle system initialization. As the system boots, kernel thread stacks are allocated before the vnode pager task is initialized and well before a paging file has been established. Thus there is a collection of kernel stacks which do not have preallocated backing store.

There are two possible solutions to this problem: statically preallocate a fixed number of startup stacks or dynamically allocate secondary storage for existing stacks once the vnode pager is available. The former solution was rejected because of the more difficult problem of selecting an appropriate number of preallocated stacks for an operating system that is intended to run across a very large range of machines and configurations.

Instead of a static solution we chose to "record" all stack allocation and free requests and "play back" these requests once the pager is ready to receive backing store requests. The vnode pager signals its readiness when swapon() calls into the kernel with stack_init_backing_store(). This routine initializes the kernel object and replays the allocation/deallocation request queue. Should an allocation request fail then the unplayed requests are left on the queue. A subsequent swapon() can then satisfy these requests. Otherwise, some of the startup stacks will still be left with no preallocated backing store. The existing stack cache lock is used to protect transitions on the record queue.

The second change was made for performance reasons. The operating system already maintains a cache of thread kernel stacks. Stacks get added to this cache as threads are created until the cache limit is reached. Therefore the allocation and deallocation of backing store is not made when the thread is created and destroyed but when a new kernel stack is acquired or released via kmem_alloc() and kmem_free(). This means that the cached kernel stacks retain their backing store preallocation even if they are not being used. A possible future performance optimization is to allocate multiple kernel stacks when the stack cache is empty to amortize the cost of the backing store allocation.

One routine, stack_free_backing(), is added to support deallocation of backing store when a kernel thread is made unswappable. This is done under the assumption that once made unswappable, kernel threads are never again made swappable. This is enforced by a call to panic() in thread_swappable(). Freeing the backing store eliminates allocation of backing store that is never used. The backing store for user threads is not freed when a thread is made unswappable.

4. Clustered Paging Design

One of our performance studies measured paging and disk I/O statistics for a system that was forced into

steady state paging behavior. OSF/1 demonstrated a larger number of paging operations than a comparable operating system under similar conditions. The major factor leading to this behavior was OSF/1's use of single page I/O operations for pagein and pageout in comparison to the multiple page operations used by the other operating system. These single page operations limit the I/O bandwidth available to clean pages and service page faults. Our solution to this problem was to "cluster" page operations, i.e., transfer more than one page at once in order to improve I/O path utilization. Implementing this required three major sets of changes:

- the backing store page allocation policy was changed so that contiguous memory object pages are stored contiguously on the disk. This *clustered allocation* allows adjacent logical pages to be transferred in a single disk operation.
- when a dirty page is selected for replacement then *clustered pageout* will also select adjacent pages in the memory object to be cleaned in the same pageout operation.
- when a page is faulted in then *clustered pagein* will read adjacent pages in the same page read operation.

4.1 Clustered Allocation and the Vnode Pager

We first describe how paging works in OSF/1 R1.0 to provide background for the R1.1 changes. The OSF/1 R1.0 vnode pager associates pages of a temporary object with separate pages of backing store. All of the backing store pages for a single object are allocated from a single paging file. Paging files may be either raw disk partitions or regular files residing in file systems. Paging to raw disk partitions is more efficient, but unused paging space on a raw partition cannot be used by other files. OSF/1 R1.0 supports multiple paging files but cannot change the association of an object with a paging file after it is established. A simple mechanism is available to influence the selection of a paging file: a paging file can be declared as preferred when the swapon command is issued to add it to the system. Preferred paging files will be used before space is allocated from non-preferred ones.

There are two steps involved in establishing an association between an object and its pages on backing store. The first step associates a data structure in the pager with the object in the VM system. The VM system initiates this step by sending a memory_object_create() message to the pager. In response, the vnode pager sets up a data structure (called a *vstruct*) to represent the object and assigns a paging file to contain its pages. No allocation of pages occurs at this time. The second step involves the actual pageout of pages. The VM system initiates this step by sending a memory_object_data_write() message to the pager in order to page out a page. At this time the vnode pager allocates a page from the previously selected paging file and the page is written to the backing store. These backing store pages remain allocated until the object is destroyed and its resources are reclaimed.

These paging mechanisms have been generalized in OSF/1 R1.1. Allocation of pages occurs in units of clusters instead of single pages to better match the multi-page pagein and pageout operations that the VM system uses in R1.1. The allocation of paging file pages has been generalized to allow pages for a single object to be obtained from multiple paging files, and the preference mechanism has been extended to improve control over the use of paging file space.

4.1.1 Paging Files

The R1.1 code allows a single VM object to be automatically backed by more than one paging file. This replaces R1.0's associations of memory objects and paging files with associations of page clusters and paging files. Creation of these cluster associations is deferred until pageout of the cluster occurs; this automatically spreads paging activity across the available paging files and adds another use of lazy evaluation to those already present in the Mach virtual memory system.

A priority scheme is used to select paging files instead of the previous binary (preferred/not preferred) scheme. This priority is set by the swapon call that initializes a paging file. Multiple priority levels for paging files are a better match to multiple levels of secondary storage performance. For example, a system could be configured to select backing storage from a ram disk, followed by a parallel disk array, single disk drives, and finally from across the network as a last resort. The paging file selection algorithm is invoked each time a page cluster needs to be allocated (as a result of pageout). This algorithm cycles through all paging files of the highest priority, in round robin order, spreading allocations across them. When there is no more space available at a particular priority level then the files at the next available level are used.

4.1.2 Clustered Allocation

One of our design goals was to limit dependencies on how the kernel implements clustered paging. In particular, our design exhibits the following flexibility:

- Pagein/out requests may exceed paging file cluster size.
- Pagein/out requests can start on any page boundary within a cluster.
- Pagein/out requests may span cluster boundaries.

The design does require that the cluster size be a power of two multiple of pages. This allows us to track paging file page allocation on a per-cluster instead of a per-page basis, reducing the amount of state and simplifying the algorithms. We don't expect this to be a significant limitation in practice.

The vstruct (vnode pager data structure that represents an object) is the primary source of information about the location of the object's pages on backing store. This structure contains the size of a cluster in pages and a map of allocated clusters. Due to the use of lazy evaluation, this map is usually sparse. Each entry in this map indicates the paging file from which the cluster was allocated, the offset of the cluster within the paging file and a bit mask of valid pages in the cluster. The bit mask is necessary to support pageout of partial clusters and eager allocation of backing storage (which results in clusters that contain no valid pages). Our current implementation packs this information into a single 32 bit field to minimize the space occupied by these data structures. The boundaries between the components of this field are variable; here is the layout we use by default:

#bits	5	1-8	19-26
field	paging file	bitmap	cluster offset

The paging file field is an offset into an array of paging file pointers. A field size of five bits limits the number of paging files to 32. The bitmap field must be sized to match the number of pages in a cluster; the current implementation allows cluster sizes ranging from one to eight pages. The remaining bits are used for the offset of the cluster within the paging file. The cluster offset in the paging file varies inversely with the bitmap size.

The flexibility and scalability of our implementation are enhanced by severely limiting the number of routines that have knowledge of this data structure. This makes it easy to adjust the boundaries among the fields (e.g., to allow a larger number of paging files). It is also straightforward to change to a new representation that uses more space (e.g., to scale to larger quantities of backing store), because the internal interfaces have been abstracted so that they do not depend on this representation.

Our current implementation uses a system-wide default for the cluster size for all paging files and objects in the system. This optimizes the use of space, but the design allows paging file cluster sizes and VM object cluster sizes to vary. The assignment of cluster size to a paging file occurs at paging file creation; it is an optional argument to the swapon command, and defaults to eight pages. In turn this default can be set by a system administrator or vendor to 1, 2, 4, or 8 pages. When paging from an actual file instead of a

paging file (e.g., to support mapped file functionality), the cluster size is set to match the filesystem block size, because this is likely to be the size of contiguous allocations in the filesystem. For example, if the filesystem block size is 8k, and the vm page size is 4k, then the cluster size will be set to two pages. This policy can be modified to depend on filesystem type.

4.2 Clustered Pagein

Clustered pagein retrieves a faulted page and some number of adjacent pages in a single read operation. Clustered pageins improve performance by taking advantage of locality of reference; there is a good chance that there will be page faults in the near future on pages close to the initially faulted page. Retrieving these pages in advance avoids the need to access disk when these faults occur. In the absence of locality of reference, the performance degradation is minor; the scarce resource in paging is disk operations, and clustering of pageins does not involve additional disk operations. The major source of performance improvement comes from amortizing expensive disk transfers over multiple pages.

No changes to the existing EMMI interfaces were required to support clustered pagein, as *memory_object_data_request()*, *memory_object_data_provided()*, *memory_object_data_unavailable()*, and *memory_object_data_error()* already support multiple pages. The implementation of these routines and related code was improved by incorporating Mach 3.0 techniques into OSF/1 R1.1. These techniques are known as "fictitious pages" and "page stealing." In addition, we retained and enhanced a performance improvement introduced in R1.0, "short circuiting."

4.2.1 Fictitious Pages

Fictitious pages are used in the fault path to improve physical memory utilization. OSF/1 R1.0 used Mach 2.5's technique of allocating physical pages in the fault path to block subsequent faults and reserve the memory that will eventually satisfy the fault (via a pagein, page copy, or zero fill, depending on the fault). The need to block faults reaching an object via different paths may require the allocation of two physical pages to satisfy a single page fault. To reduce this use of physical memory R1.1 employs Mach 3.0's technique of allocating fictitious pages as placeholders in the fault path. A fictitious page consists of just a vm_page data structure, without any associated physical memory. Using fictitious pages in the fault path defers allocation of physical memory until it is actually needed to satisfy the fault. This reduces the use of physical memory at the cost of a small increase in the complexity of the resident page management code in the virtual memory system.

4.2.2 Page Stealing

Page stealing logic transfers pages directly from the pager to the destination object without performing a copy. To use page stealing effectively, the pager must allocate its own buffers for retrieving data, and allow the data to be deallocated as a side effect of the pagein. If the pager wishes to retain the data in its address space after the pagein, then the pages cannot be transferred and must be copied. The actual data transfer swaps the pages for the fictitious pages serving as placeholders, and frees the fictitious pages. A new interface, *memory_object_data_supply()*, is used in place of *memory_object_data_provided()* to enable page stealing. Due to a limitation of the Mach 2.5 version of MiG used in OSF/1, memory_object_data_supply() always deallocates the paged-in buffer; memory_object_data_provided() is still available for pagers that wish to retain copies of paged-in data.

4.2.3 Short Circuiting

The short circuiting optimization allows faulting threads to directly execute the vnode pager's pagein code. This addresses a major problem with older versions of Mach (including that used as the base for R1.0), namely that the pagein path always makes a copy of the page being paged in. Short circuiting

eliminates this copy by paging in directly to the page reserved by the page fault logic. Short circuiting also eliminates the exchange of messages and context switches required to invoke the vnode pager, and enhances throughput by allowing non-vnode pager threads to execute the pagein code. Although page stealing eliminates the page copy (reducing the performance advantage of short circuiting), we chose to reimplement short circuiting in R1.1 to retain its other advantages.

Short circuiting is not used for pageout because the Mach pageout daemon is a single thread. By transferring responsibility for performing the pageout, a single pageout thread can keep multiple pager threads (and hence paging files) busy, achieving higher bandwidth than would be possible with a single thread. There is also an important robustness advantage to this transfer. Some paging files (e.g., those accessed over NFS) are prone to temporary inaccessibility. By delegating a different vnode pager thread to actually access the paging file, the pageout thread is insulated from this type of problem.

4.3 Clustered Pageout

Combining multiple pages in a single (clustered) pageout operation can save write operations to backing store. Clustering combines adjacent dirty pages with the original candidate page for pageout and writes the collection to backing store in a single operation. After this write, only the original candidate would normally be freed; the other pages retain their positions in the pageout queues. If these pages remain clean until they become candidates for page replacement, then they can be freed instead of being written; this saves write operations. As in the pagein case, the scarce resource is disk operations, so the added overhead of writing a cluster instead of a single page is minor even if the additional pages become dirty and require their own write operations.

It was much more difficult to implement clustered pageout than pagein. This was caused by the complexity of the Mach pageout path and by assumptions in both the pager and pageout daemon that pageout operations only involved single pages. A major contributor to the complexity is the asynchronous nature of pageout; this required additional state to track pageouts in progress for both the kernel and pager. We also found it necessary to introduce a new method for pageout of dirty pages, cleaning in place, and discovered that careful coding was required to ensure that the interactions between clustered paging and the page queues did not disrupt the approximate LRU algorithm used to select pageout candidates.

4.3.1 Cleaning in Place

It is necessary to first understand how OSF/1 R1.0 pageout works before explaining the need for cleaning in place. When the system needs to move pages to the free list the pageout daemon selects the oldest page from the inactive queue for replacement. If the page has been referenced while on the inactive queue then it is reactivated. If the page is not dirty, then it is immediately placed on the free list. Otherwise, the page must be cleaned (paged out) before it can be released.

Pageout is implemented by moving the page to a new VM object. This page is temporarily replaced in its original object by a busy fictitious page to prevent faults on this page from causing pageins before the pageout message is sent. The real page is moved into a new VM object, created for the purpose of this pageout, and sent to the pager. The fictitious page is then released. The new VM object is used primarily to protect Mach from recalcitrant user-written pagers. The system's default pager backs the new object, and can write the pages to its backing store if a user-written pager is unwilling to do so. This level of protection is not necessary for interactions with the default pager itself.

This replacement mechanism is not appropriate for clustered pageout. Making the entire cluster of pages inaccessible for the duration of the pageout has a number of undesirable effects. It increases the cost associated with these pageouts, and also effectively increases the system's page size. The major problem is that this risks selecting the wrong pages for pageout since other pages in the cluster could be much more active than the original candidate selected for replacement. Instead, we changed the implementation of

pageout in R1.1 to clean pages in place without moving them into a separate VM object. The adjacent pages in the cluster are left in the original VM object and remain accessible (the target page also remains in the VM object, but becomes inaccessible because in order to guarantee that it will be successfully cleaned and freed). All of these pages are marked as unmodified before the pageout is initiated; the alternative, waiting until pageout completion to do this, risks losing modifications made while the pageout was in progress.

Cleaning in place cannot be used with untrusted pagers because it eliminates the temporary object backed by the default pager. Our implementation does not perform clustered pageout to an untrusted pager, but instead uses the existing Mach single-page mechanisms to retain protection from recalcitrant pagers. An additional test has been added to the pageout code that allows the vnode pager to be considered trusted for both internal (default pager) and permanent (file) objects.

Two additional flags were added to the VM page structure to detect intermediate states in the new pageout process, the "cleaning" and "pageout" flags. The "cleaning" flag marks a page for which a pageout is in progress. This avoids initiating duplicate pageouts on these pages, and allows the operation that completes pageout to detect that it is manipulating the correct pages. A new call, *memory_object_data_write_completed()*, was created so that the pager could inform the VM subsystem of pageout completion, so that the "cleaning" flag can be cleared for the pages involved. The "pageout" flag identifies a target page for pageout (this is a page that has been selected for cleaning, and is therefore not on the pageout queues). The memory_object_data_write_completed() operation frees such pages if they are clean; otherwise it returns them to the pageout queues. If a non-target page (adjacent to the original target page in the cluster) reaches the end of the inactive queue before its pageout completes, its "pageout" flag is set to mark it as a target page. The cleaning flag is also used to synchronize cleaning in place with copy-on-write.

The pages in the cluster cannot be deallocated while they are in the process of being cleaned. To prevent this the pageout daemon takes a paging in progress (software) reference on the object's data structure for each page being cleaned. These references are released as part of the memory_object_data_write_completed() call. vm_object_deallocate() synchronizes with the release of these paging in progress references.

4.3.2 Effect on LRU Approximation

Mach uses the active and inactive queues to approximate an LRU page replacement algorithm. The pmap module is responsible for implementing a reference bit per physical page that can be cleared and checked. The pageout daemon clears this bit when placing a page on the inactive queue, and checks it when the page reaches the end of the queue. If the bit is set, the page is returned to the active queue instead of being paged out or freed. This reference bit should not be confused with the modify bit that is set only by writes to the page.

Mach's association of the reference bit with the physical page instead of the virtual to physical mapping (e.g., as in 4.2BSD) is a potential source of problems. Systems that associate the reference bit with the mapping can distinguish between accesses that use different mappings; this allows such systems to ignore references made by their pageout mechanisms by using a separate mapping. Investigation revealed that the vnode pager in OSF/1 R1.0 would always set the reference bit in the process of paging out the page. This is because that pager wires and unwires pages, and may actually reference the data (typically on machines that lack DMA hardware).

This behavior seriously distorts the LRU approximation. Consider the situation in which a group of pages is paged in and modified, then not used for a long period of time. When the first of these pages reaches the end of the inactive queue, it will be selected for replacement, and cleaned. The adjacent pages (which are near the end of the inactive queue) will also be selected to be cleaned as a cluster. At the completion of the cluster write, the target page is freed, but the other pages are reactivated (because the pager set their

reference bits). Rather than being replaced soon (since they are inactive) they are now the least eligible pages in the system for replacement. To prevent this behavior the vnode pager was modified to not wire the pages into its address space (this also shortens the pageout code path).

This has repercussions for those doing OSF/1 ports. Their ports must be examined to ensure that the act of performing a disk write from the memory region set up by the pager does not cause any fault on the pages or any change to the reference bits implemented by the pmap module. Otherwise clustered paging will distort the LRU page replacement algorithm.

5. Swapping

When the total demand for page frames, i.e. the system-wide working set, exceeds the total available resident memory space then the amount of useful work accomplished between page faults falls. This causes an increase in the amount of system resources dedicated to moving pages between backing store and primary memory, with a corresponding decrease in available CPU cycles to perform useful work. As the problem gets worse, it is even possible to fetch pages into main memory and then discard them prior to their being used even once. This awkward degradation of demand-paged virtual memory systems is called thrashing. A thrashing system is spending too large a fraction of its available resources on paging, and too small a portion on actual computation.

OSF/1 R1.0 makes no attempt to manage the demand for page frames, and is therefore subject to thrashing under even moderate loads. To correct this, we have implemented swapping: the detection of excessive page demand and the selection and suspension of tasks to reduce that demand. This section describes how the Mach based OSF/1 operating system was modified to add swapping. The discussion is divided into sections on swapping policy and mechanisms. Much of the motivation for this work arose from the swapping implementation in 4.3BSD Unix.

5.1 Swapping Policy

Swapping addresses the following problems in the 4.3BSD Unix system:

- The system page map becomes fragmented, preventing processes from allocating space for page tables.
- Processes are inactive for more than 20 seconds, but continue to consume page table resources.
- The paging rate is excessive, so the pageout daemon cannot clean pages fast enough to meet demand.

The first two problems do not exist in OSF/1, because the Mach VM system uses different data structures and allows resources (e.g., page tables) to be reclaimed. Fragmentation is prevented by the use of appropriate resource management techniques in the pmap module. (Note: it is possible to write a pmap module to create this problem but a well written pmap module will not fragment.) The consumption of page table resources by idle processes is prevented by the thread swapper. The thread swapper scans the all threads list looking for threads that have been idle for at least 10 seconds. Such threads have their kernel stack unwired, and the pmap module is invoked to reclaim the thread's page table resources.

The problem that was not addressed in R1.0 is excessive demand for physical memory. When the total working set of all active tasks exceeds the amount of memory available, the system needs to prevent some tasks from running. Two policy questions must be answered when swapping becomes necessary: which tasks should be swapped out and when should swapped tasks be swapped back in?

Our swapout decisions are based on the algorithm used in 4.3BSD Unix. That algorithm maintains resident set size and memory residence time information for each process. Of the four processes with the largest resident set size, the one that has been resident in memory the longest is chosen as the victim to be swapped out. We chose this algorithm based on its simplicity and the experience with its use in 4.3BSD

Unix.

The OSF/1 R1.1 pageout daemon initiates task swapping when the free page count remains low despite the pageout daemon's best efforts. Paging activity in the pageout daemon is controlled by two thresholds on the amount of free memory, a minimum (typically 1% of memory) and a target (typically 1.25% of memory). The pageout daemon is invoked when the amount of free memory drops below the minimum threshold, and remains active until the target threshold is reached. The higher target threshold improves the utilization of the pageout daemon by invoking it less often to do more work each time. Task swapping is invoked when the amount of free memory remains both below the minimum threshold for 5 seconds and below the target threshold for 30 seconds. These delay times were also taken from 4.3BSD Unix, but are easily modified.

The task swapper chooses a task to swap out by using the 4.3BSD Unix swapout algorithm. Of the largest tasks in the system, the one that has been resident the longest is selected for swap out. The current resident set size of the task is used as an estimate of the number of pages that will be freed by this operation. (A better estimate would be to use the non-shared resident set size, but this number is currently not maintained by the pmap modules.) Tasks swapped by this swapper are *involuntarily* swapped; they will not be permitted to run again for some period of time. A task can also be swapped out *voluntarily* when the thread swapper swaps all of the task's threads.

There are two ways a non-resident task can be swapped in: a voluntarily swapped task will be swapped in when any of its threads are awakened as a side effect of swapping in the thread. An involuntarily swapped task will become eligible for swapin when the memory shortage has been relieved sufficiently to allow the task to make progress (based on the task's resident set size at swapout time) or when the task has been non-resident for longer than an adjustable threshold. If swapping in a task to prevent starvation, the swapper may choose to swap out another task first. A list of all involuntarily swapped tasks is maintained to implement this time based swapin algorithm.

When implementing these swapping policies we made a strong effort to separate policy and mechanism. We recognize that there are a wide range of possible swapping policies and that system vendors will will need to adjust or replace this policy when tuning OSF/1 for their particular systems.

5.1.1 Additional Statistics

Some statistics needed to be added to the Mach kernel in order to support the policy decisions. These statistics had to be added to the thread, task, and Unix data structures.

The kernel already tracks a set of system-wide virtual memory statistics such as pageins, copy-on-write faults, and page reclaims. To supplement the existing statistics an events_info structure was added to the thread and task structures. The thread statistics are incremented when the kernel updates the global statistics. Per thread statistics are accumulated in the per task statistics at thread exit. In addition, the cumulative statistics for all terminated child processes of a (Unix) process is recorded. This data is stored directly in the u.u_cru structure in the U-area (utask). This information is now available to user space via the rusage structure and the task_info() and thread_info() Mach calls.

5.2 Swapping Mechanism

The swapping mechanisms implement swapping policy decisions. The swapout mechanisms are responsible for suspending the execution of victim tasks and reclaiming appropriate resources from them. The swapin mechanisms are responsible for allowing task execution to resume and resources to be restored. This section describes the implementation of these mechanisms in OSF/1 R1.1. We begin with a brief description of the virtual memory data structures that are involved.

5.2.1 Virtual Memory Data Structures

This is a brief summary of the Mach virtual memory (VM) data structures. The reader should already be familiar with Mach VM but this section is provided as a convenient reference. An understanding of the relationships between VM data structures is necessary in order to explain the swapping mechanisms.

Although the figure shows the most common relationships between VM components it is by no means complete. One could add to the complexity by including additional tasks pointing to the sharing map, a more complicated shadow object chain, and copy objects. However, this diagram provides enough support for the following discussion.

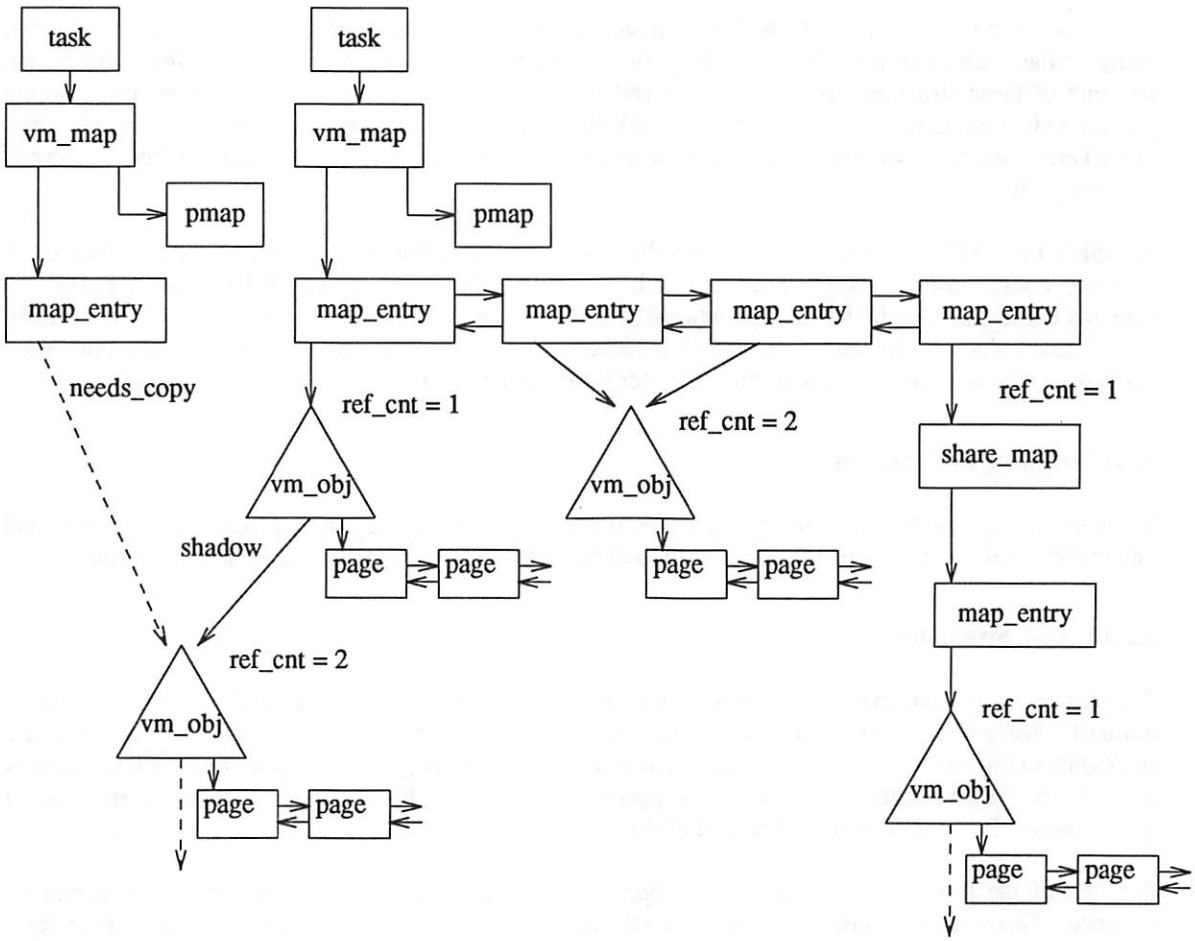


Figure 1. Relationship between VM data structures

A task's virtual address space is described by a *vm_map*. The vm_map has access to the physical mapping for that task and the linked list of map entries that describe each virtual memory region. Each *map_entry* points to either a *vm_object* or a *share_map*. The vm_object represents the set of pages that are mapped to the corresponding region. The vm_object keeps a linked list of the region's resident pages, a port that is used to communicate with a memory manager for non-resident pages, and a possible pointer to other vm_objects which describe additional pages within the region. A map_entry can also point to a *share_map* in the case where multiple tasks are sharing memory. The share_map points to its own linked list of map entries and their vm_objects.

Sharing of physical pages can occur in a variety of ways with these data structures. A sharing map causes a portion of address space to be shared among two or more tasks. An object can be independently mapped

into the address space of more than one task. Often the sharing is indirect due to the use of virtual copy optimizations; objects mapped into two different tasks may be copies of a single underlying (or shadowed) object. Multiple levels of such shadowing are possible (and occur in practice).

5.2.2 Reclaimed Resources

We chose to reclaim only a critical subset of the resources that could be reclaimed from a swapped task. Among these resources that could be reclaimed are: task state (including VM structures), thread state, thread stacks, Unix process state, task resident pages, and the task's page table (pmap). We chose to focus on thread stacks, the task's resident pages, and the task's pmap. This selection gave us the largest benefit for the initial swapping implementation.

One reason for not attempting to reclaim the smaller structures associated with a swapped task is that many of them consume a fraction of a page that is shared with other unrelated structures. Writing the contents of these structures to backing store and freeing them back to their respective allocator would provide only a marginal benefit because it is unlikely to free the entire page. Beyond this, many allocators in the kernel retain a pool of available pages or structures instead of aggressively releasing free memory to the page pool.

In OSF/1 the BSD style U-area has been split into two pieces, the utask and the uthread. The utask structure holds those task specific fields which were once part of the U-area and the uthread structure contains the thread specific fields. The uthread is stored in a thread's kernel stack so it will automatically be reclaimed when the thread's kernel stack is reclaimed. The utask structure and the logically connected per-process file table are very good candidates for future reclamation.

5.2.3 Resource Reclamation

When examining the VM structure relationships diagram one can identify three levels: the task, map, and vm_object levels. Changes had to be made to each of these levels in order to reclaim a task's resources.

5.2.3.1 Task Swapping

The changes at the task level consist of some minor additions to the task structure and the addition of three routines. We added a state variable, swap_state, to the task structure. This has six states that are analogous to thread swap states. A queue chain is used to link together all swapped tasks. A timestamp is added to record the last time the task was swapped in or out. When the task is swapped out the resident set size is saved. This size is later used as part of the swap in criteria.

The first of the three new routines, *task_swappable()*, is used to indicate that a given task cannot be swapped. This is used for tasks associated with the pageout path. Among the non-swappable tasks in R1.1 are the first task, the vnode pager tasks, the device pager, the exception handler, and the NFS client tasks.

Task swapping is implemented by a task swapping daemon. This is similar to the pageout daemon except that it swaps tasks instead of paging out pages. When this daemon finds an appropriate victim task to swap, it calls *task_swapout()*.

```
void
task_swapout(task_t task)
{
    record resident set size in task structure

    task_hold(task);        // suspend all threads
    task_dowait(task);      // wait for all threads to stop

    add task to list of non-resident tasks

    // start swapping threads
    thread = task->thread_list;
    while (thread)

        // mark thread swapped, swapping out.
        thread_swapout(thread);
        thread = thread->thread_list;
}
```

Figure 2. task_swapout pseudocode

There are three steps involved in swapping a task, suspending its threads, swapping them, and swapping the task's memory. Task_swapout first suspends all threads with the task_hold and task_dowait routines. (These routines were already present in R1.0.) These increment the suspend count for all the threads in the task and then wait for them to actually suspend. Task_swapout then swaps out all the threads. This unwires the threads' kernel stacks, making the eligible for pageout. Thread_swapout also tracks the number of swapped threads in a task; when the last thread is swapped, the pmap module is invoked to reclaim the task's page tables, and vm_map_res_deallocate() is called to swap out the task's resident pages. This routine is discussed in detail in the next section.

The third new routine is *task_swapin()*, the inverse of task_swapout(). To swap in a task, the task is marked not swapped and the suspend count on all threads is reduced via task_release(). Task_release() then makes all threads with a zero suspend count runnable. When the first thread becomes runnable the scheduler will make the task's map resident by calling vm_map_res_reference().

The swapin time is saved in the task structure so that the swapout policy can calculate the residence time (and ensure that a swapped in task remains swapped in for some minimum time period). Swapping in a task does not involve explicitly bringing pages from secondary store into main memory. The task will begin faulting in its working set as soon as it is made runnable. Clustered paging improves the efficiency of restarting a swapped task.

5.2.3.2 Map and Object Swapping

This section covers the swapping of the lower layers in the VM system, the map and object layers. We discuss these layers as a unit because similar mechanisms are used for both. In reading this discussion it is important to understand that the the map and object data structures themselves are not swapped. Rather these structures are traversed and modified to allow the contents of resident pages to be swapped out and the page tables to be reclaimed. Actually swapping out the map and object data structure is a topic for potential future work.

Traversing the complex relationships between the VM structures is the most difficult aspect of designing the OSF/1 swapping mechanisms. The design is greatly simplified by relying on the Mach techniques of reference counts and lazy evaluation.

A resident count field is added to both the vm_map and vm_object structures. Thie field parallels the reference count; every holder of a reference on the data structure (i.e., has incremented the reference count) may optionally hold a resident reference (i.e., has incremented the resident count). The map or object is considered swapped in only when the resident count is non-zero (i.e., 'swapped out' maps and vm_objects always have zero resident counts). The act of swapping a task's pages consists of decrementing the resident count on the task's VM map that corresponds to the reference held by the task data structure. When the resident count becomes zero, the algorithm proceeds to the next layer and repeats; each map and object structure in the next lower layer has its resident count decremented. This recursion terminates when a non-zero resident count (after decrementation) is found or when the bottom is reached. When a vm_object's resident count reaches zero, those pages are not in use by any active thread are deactivated. The table below summarizes the R1.1 routines which affect the vm_map and vm_object resident counts.

Map Swapping	res	ref	Object Swapping	res	ref
vm_map_create	1	1	vm_object_allocate	1	1
vm_map_res_deallocate	-	nc	vm_object_res_deallocate	-	nc
vm_map_deallocate	-	-	vm_object_deallocate	-	-
vm_map_res_reference	+	nc	vm_object_res_reference	+	nc
vm_map_reference	+	+	vm_object_reference	+	+
vm_map_swapin	+	+			

- == decrement 1 == assign one
+ == increment nc == no change

Figure 3. Resident and Reference Counts

The resident and reference counts match unless swapping has occurred. This is because the routines that change only one count but not the other (the *_res_reference and *_res_deallocate routines) are only called by swapping code. The normal operations that create, deallocate, and reference maps and objects make the same changes to both counts.

The swapping algorithm implements a conservative approach to swapping shared memory. Such memory is swapped when the last task sharing it is swapped. This behavior depends on the use of references by the Mach kernel. In a quiescent state (no operations in progress) a top-level (task) VM map will have exactly one reference, the reference held by the task data structure. All other sharing maps and objects will have one reference from each higher level object that points to them (i.e., shares their contents). The maintenance of resident counts in each data structure ensures that a resident page can only be forced out by swapping when all tasks into which it may be mapped are swapped. If any such task is not swapped, its VM map will have a non-zero resident count, and hence all sharing maps and objects below that task's VM map will also have non-zero resident counts. This bookkeeping for shared regions is the source of most of the complexity in the swapping implementation, but we felt that taking this approach to shared memory was important. For example, the C library is shared in OSF/1; it is important that the library remain swapped in while any task is using it.

A consequence of this is that any operation that takes or releases a map reference must be prepared to cause the corresponding swapin or swapout. The fact that these are blocking operations causes a difficulty with the code that manages vm_maps. Map references are often taken in circumstances where the code is not prepared to block; these circumstances are always ones in which the code already holds a reference on the map and is cloning the reference. This cloning of a reference can never cause a swapin. To identify such clonings and make this assumption explicit, we use different routines for this case and the case of taking a reference that can cause a swapin; both routines have the effect of incrementing both the resident and reference counts. We used vm_map_reference for the cloning case (swapin not possible) since this corresponds to most existing uses of that routine. For the cases in which taking a reference may cause a

swapin, the new *vm_map_swapin* routine is used. An important example of such a case is the reference taken by convert_port_to_map() when the invocation of a Mach VM operation crosses a task boundary (i.e., the invoking thread is not in the task affected by the operation). This reference ensures that a VM map is considered swapped in while an operation is in progress on it.

6. Conclusion

One of the more important conclusions of our work is that the Mach virtual memory subsystem is flexible enough to allow these large functional changes without requiring major overhaul of the underlying design. Also, we feel that these robustness and performance changes to the virtual memory subsystem have greatly improved the commercial viability of the OSF/1 operating system. The robustness changes, deadlock removal, and eager allocation of kernel stacks, have increased system availability and preliminary performance measurements of the clustered paging changes show that larger numbers of pages are being transferred with each I/O operation, as expected. Complete performance numbers were not available at the time this paper was written. We expect to have performance results at the time of the conference.

References

[1] David L. Black, Avadis Tevanian,Jr., David B. Golub, and Michael W. Young, "Locking and Reference Counting in the Mach Kernel", Proceedings of the 1991 International Conference on Parallel Processing, August 1991, pp. II-167 - II-173.

[2] S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman, "The Design and Implementation of the 4.3BSD Unix Operating System", Addison-Wesley, Reading, MA (1989).

[3] R. Rashid, A. Tevanian, Jr., M. Young, D. Golub, R. Baron, D. Black, W.J. Bolosky, and J. Chew, "Machine-Independent Virual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", IEEE Transactions on Computers, Vol. 37 No. 8 (August 1988), pp. 896-908.

[4] Avadis Tevanian, Jr., "Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach", PhD thesis, CMU-CS-99-106, Dept. of Computer Science, Carnegie-Mellon University, December, 1987.

[5] Jim Van Sciver, Ping Wang, "OSF/1 (1.0.1s+) System Memory Usage", OSF/1 Release 1.1 Engineering Note, Open Software Foundation, May, 1991.

[6] Michael Wayne Young, "Exporting a User Interface to Memory Management from a Communication-Oriented Operating System", PhD thesis, CMU-CS-89-202, Dept. of Computer Science, Carnegie-Mellon University, November, 1989.

[7] M. Young, A. Tevanian, Jr., R. Rashid, D. Golub, J. Eppinger, J. Chew, W.J. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System", Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Vol. 21, No. 5 (November 1987), pp. 63-77.

[8] "The Design of the OSF/1 Operating System", Book in Progress, Open Software Foundation.

[9] "Guide to OSF/1: A Technical Synopsis", O'Reilly & Assiciates, Inc., 1991.

Parallelizing Signal Handling and Process Management in OSF/1 *

Don Bolinger *bolinger@encore.com*
Shashi Mangalat *shashi@encore.com*

Mach Operating System Project
Encore Computer Corporation
257 Cedar Hill Street
Marlborough, MA 01752

Abstract

Release 1.0 of the OSF/1 operating system, despite its high degree of parallelization, left several dozen system calls unparallelized. The most important subsystems not converted were process management and signal handling. This paper describes the project to make these subsystems multiprocessor-efficient, and to make their system calls usable within multi-threaded tasks. After presenting background on OSF/1 and on the relevant system calls, we describe the general approach and specific changes we adopted for the parallelization, and for the adaptation of Unix process-oriented abstractions to the multi-threaded programming model of OSF/1. After providing rationales for our most important choices, and comparing them to a few discarded alternatives, we look at how some common operations are implemented in the resulting kernel, examining the resolution of races and other synchronization problems introduced by our changes. Finally, we present data on performance improvements introduced by the project, and indicate a few possibilities for useful future development.

1 Introduction

The OSF/1 operating system consists of two major parts. The core of the system is the native Mach kernel, which provides the mechanisms needed for operation in a distributed environment using either uniprocessor or multiprocessor (MP) hosts. In addition, OSF/1 (like Mach itself) contains code to emulate the programming (system call) interface of a BSD Unix operating system, in this case the 4.3BSD-Reno release.

The native Mach kernel, of course, is fully parallelized for use on tightly-coupled shared-memory multiprocessor (SMP) architectures. The original 4.3BSD-Reno code used for Unix emulation, however, was designed to run on a uniprocessor, and uses interrupt-level (spl) based synchronization mechanisms that fail in an SMP environment. Of this code, the most heavily-used subsystems — notably the networking and TTY subsystems and all filesystem code — were fully parallelized before Release 1.0 [9, 2]. The remaining emulation code, however, is unparallelized, and must run on a single master processor, which is permanently designated at boot time. Use of this code therefore

*Multimax, UMAX4.3 and UMAXV are trademarks of Encore Computer Corporation. Unix is a trademark of AT&T Bell Laboratories. OSF/1 is a trademark of the Open Software Foundation.

obliges a thread to wait for and execute on the master, an operation implemented by the function `unix_master`, and which we will refer to by that name.

A major goal for Release 1.1 of OSF/1 was to eliminate as many of the remaining uses of `unix_master` as possible, since they degrade system performance not only by serializing the execution of unrelated threads, but also by forcing otherwise-needless context switches to the master CPU. Of the kernel code that had not already been parallelized, the Unix system calls relating to process management and signal handling, along with the kernel code involved in signal generation and delivery, were seen as the most urgent candidates for conversion. Some of these calls (like *sigvec* or *sigaction*) are used very frequently. Others (like *fork* and *exit*), though less common, take a very long time to execute.

Parallelizing these system calls, and the associated signal processing code, involved both making them MP-efficient, and making them thread-safe, that is, suitable for use in a multi-threaded task. Thus we enabled the calls to be run on any processor in an SMP system by adding appropriate locks and other synchronization mechanisms. After ensuring that the calls obeyed the appropriate specification (BSD or POSIX) in single-threaded use, we then implemented correct behavior in multi-threaded tasks, as well.

We did not seek to provide conformance with any particular model of multi-threaded task behavior, but rather tried to emulate single-threaded semantics as closely as possible. Where changes in multi-threaded behavior were unavoidable in achieving this goal, we used a recent draft document (draft 5) of the POSIX 1003.4a (Pthreads) working group as a guide in our own implementation.

Though references exist in the literature to parallelization efforts involving BSD-derived Unix kernels [2, 11, 6]), and to process migration schemes to enhance the performance of process manipulation on MP architectures [5, 4], we are unaware of any detailed description of the problems involved in parallelizing Unix process management for use in an SMP environment, and of how to resolve these problems effectively. We hope that sharing our experience may be of use to others faced with similar tasks, in providing some data to simplify the undertaking.

2 Background

2.1 Related Unix system calls

The main process management system calls treated by this project were the classic Unix quadrumvirate: *fork*, *exec*, *wait* and *exit*. We assume that the reader is familiar with them, so we will not explain their operation here. As it happens, many other, less important system calls in this area (like *getpriority* and *setpriority*) were also parallelized in the course of our work, but their cases are not of sufficient interest to merit inclusion here.

It may be more useful, however, to recap the Unix signal mechanism and a few associated abstractions before going further.

2.1.1 Signals

A signal is a software interrupt, sent to a given active process to indicate some exceptional condition (either hardware- or software-related). Though they were initially meant to provide a mechanism for explicit manual intervention in resolving errors (see [8] for an example of how primitive early signal implementations were), signals in modern-day Unix systems are used for many purposes (look at [1, 7], for instance, for a full description). A few uses of signals that were particularly important to our project are:

- to indicate a resource limit has been exceeded
- to suspend or unsuspend a process
- to indicate that a process without access to the terminal wants to do input or output

A signal may be generated either by a user application, via the *kill* system call, or within the kernel. Signal generation does not directly change the execution state of the targeted process — instead the signal is recorded (marked as pending) for receipt by the process at a convenient point.

With some exceptions, a process may choose to ignore a signal, to let it retain its default behavior, or to associate a handler function with it. By default, most signals terminate the targeted process, though a few by default are ignored. A process may usually choose to mask a signal temporarily without affecting how the signal will later be handled — the signal will cause the action specified as soon as it is unmasked.

Three concepts associated with signals are particularly relevant to the changes we made in the course of our project. These are process groups, job control, and sessions.

2.1.2 Process groups

A *process group* is a mechanism for identifying a set of processes that can be signalled as a unit, as a way of ensuring that all processes in the group receive a given signal more-or-less simultaneously. (Obviously, each process in the group can still elect to handle it differently.) The kernel never changes a process' group internally — this is done only by an explicit system call. The most common use of process groups is by shell programs, which generally put each command line entered by the user into its own group.

Any signal can be posted to a process group — one common user action is to issue a `SIGSTOP` or `SIGKILL` to a group in order to suspend or terminate its execution. Within the kernel, some of the most important uses of process groups are in the terminal handling (TTY) code. This code uses process groups as the basis for multiplexing access to a terminal between potentially many different groups. This multiplexing is known as *job control.*

2.1.3 Job control

Each command line executed by a shell that implements job control is known as a *job.* The shell process itself can also be considered to be a job, distinct from any other ones. At any given point, a single job has control over input from or output to the user's controlling terminal — this job is the *foreground job*, and its group is the *foreground process group.* If a user executes commands sequentially, without any manual signal generation, then foreground status alternates between the shell (when the user is entering a command), and each command line executed.

When commands are run asynchronously — that is, "in the background" — they do not have foreground status. If such a command tries to interact with its terminal, it will be suspended, and will not continue until the user brings it to the foreground (by instructing his shell to make it the foreground job). Similarly, a user can explicitly put the current foreground job into the background by suspending it, then continuing it, disassociated from terminal.

2.1.4 Sessions

Intuitively speaking, a *session* consists of all of the processes started during a given login session. Naturally, therefore, a session may contain several process groups, which vary over time. Generally,

a session is associated with a single controlling terminal, access to which is given to one process group (or job) at a time, using the shell facilities described above.

2.2 Synchronization overview

The native Mach kernel and the parallelized sections of the emulation code rely on two varieties of multiprocessor lock to synchronize operations between different processors. The first type of lock, called a *simple lock*, is implemented as a spin lock. A thread waiting to acquire such a lock will spin in a tight loop till it gets the lock.

The other style of lock is a blocking lock. A thread waiting to acquire this kind of lock will be blocked (i.e., will no longer execute) until the lock becomes available. A blocking lock may be used either as a *read/write lock* or as a *mutual exclusion (mutex) lock*. In the first case, multiple readers, but only one writer, may hold the lock at any given time. In the second case, only one thread may ever hold the lock at a time.

Our goals in applying these primitives can be summarized as follows. Predictably enough, some of them are mutually-conflicting.

- Minimize the scope of source-level changes.
- Make the changes automatically adaptable to different hardware configurations (e.g., uniprocessor vs. MP).
- Minimize dynamic lock scope, as by using reference counts.
- Maximize parallelism in application-visible interfaces, as well as internal to the kernel, on behalf of multi-threaded tasks.

In particular, the great majority of the synchronization mechanisms we added for use on an MP system are not needed (and are not compiled into the kernel) in ordinary uniprocessor configurations — where "ordinary" means that the user has not explicitly requested the inclusion of MP locks for debugging purposes. Details on whether a given lock disappears in uniprocessor kernels will be given when it is introduced.

3 Process and Signal Management in OSF/1.0

Among the main process management system calls treated by this project, only *exec* was already MP-efficient in OSF/1.0. Even this call required some changes for safe use in multi-threaded tasks — see Section 4.1.

In the area of signal handling, OSF/1 provides the full set of 4.3BSD-derived system calls, as well as the system call interface required by POSIX. None of these calls was parallelized in OSF/1.0, nor was the related signal code internal to the kernel. Making this subsystem MP-efficient involved changing the system call entry points involved, modifying the signal generation (`psignal`) and delivery code (`issig`, `psig`, `sendsig`), and also making excursions into the TTY subsystem and the clock interrupt handler.

Since OSF/1 provides for multiple threads of execution within an emulated Unix process, OSF/1.0 already contains several internal modifications to conventional (single-threaded) signal handling, designed to make signal handling safe in multi-threaded processes. (Most of these, and in particular the two that we will discuss here, were implemented by David Black, currently of the OSF Research Institute, for an early version of Mach.)

3.1 Signal typing

The first of these modifications is that signals are divided into per-thread and per-process types. A per-thread signal type is one directly caused by the execution of a given thread (like `SIGILL` or `SIGSEGV`). It corresponds to an exception, and is delivered synchronously to the thread in question. A per-process signal is one not caused by the execution of the targeted thread, and is delivered asynchronously to the "first" (i.e., oldest active) thread in the process.

3.2 Signal delivery in multi-threaded tasks

Another feature present in OSF/1.0 ensures that all threads in a multi-threaded task interact properly with the *exit* and *ptrace* system calls — that is, that they are properly stopped and restarted (where appropriate) when a process is being traced, and are cleanly terminated when it exits. This feature is based on a set of multi-threaded signal delivery macros, organized as follows.

The macros are based on a simple lock, called `p_siglock`, and on two other pieces of process state: a flag, called `p_sigwait`; and a thread pointer, `p_end_thread`. Together, these fields define the following states:

locked – `p_siglock` acquired. A thread is in this state only for short periods, in order to query or change the state of the other fields.

unlocked – `p_siglock` not acquired, `p_waiting` and `p_sigwait` zero. Process is not being traced, and no thread is exiting.

waiting – `p_sigwait` non-zero. A thread has already suspended the current task on behalf of a parent process that is tracing the current process using *ptrace*. No further signal processing may occur till the parent continues execution of this process.

exiting – `p_end_thread` non-zero. The thread indicated in `p_end_thread` has begun to exit, and no further signal processing should occur.

The following macros in OSF/1.0 use the above state to provide multi-thread synchronization:

`sig_lock_or_return` takes `p_siglock` to check whether current state is **waiting** or **exiting**. If not, control passes out of the macro with the lock still held. If so, the macro releases the lock and blocks or terminates the current thread as appropriate.

In reality, this macro merely marks the thread so that it will be blocked or terminated on its return to user mode, then forces a return from the function containing the use of the macro. It thus assumes that that no unintended thread manipulations will intervene before the thread is blocked or terminated. All current uses of the macro are in system call entry points, so no significant kernel processing occurs between the return of the containing function, and the actual change to the thread state.

`sig_lock_to_wait` assumes that `p_siglock` is locked. It asserts `p_sigwait`, then releases `p_siglock`. Used to await continuation by parent when process is being traced with `ptrace`.

`sig_wait_to_lock` assumes that `p_sigwait` is asserted. It acquires `p_siglock`, then deasserts `p_sigwait`. Used to co-ordinate the continuation of all threads when parent requests it.

`sig_lock_to_exit` assumes that `p_siglock` is locked. It registers the current thread as **exiting**, then halts all other threads.

4 Changes to Provide Multi-Threaded Correctness

As noted above, OSF/1.0 already contained some significant changes to ensure the predictable operation of multi-threaded tasks. In this section, we describe how we modified these to apply uniformly to the full process management and signal handling subsystem.

4.1 Multi-threaded process manipulation

First off, we extended the `sig_lock` macros described in Section 3.2 so that they applied to all process manipulation system calls.

As we've seen, the existing macros caused all threads to halt when any one thread called *exit*. We extended this mechanism so that calls to *exec* would have the same effect. We also defined an interface between this mechanism and *fork*, such that any *fork*'s in progress would complete (but no new ones could start) after either *exec* or *exit* had been called. A further description of this interface is given in Section 6.2.

Finally, in order to make multi-threaded signal handling more efficient, we introduced two macros, to repackage the transition between the **unlocked** and **waiting** states:

`sig_wait_lock` – invoke `sig_lock_or_return`, then assert `p_sigwait` and release `p_siglock`.

`sig_wait_unlock` – acquire `p_siglock`, deassert `p_sigwait`, then release `p_siglock`.

These macros avoid prolonged busy-waiting to acquire `p_siglock`, which could otherwise occur at a few points in the signal delivery code. Instead, a calling thread will either acquire the lock and put the process into the **waiting** state, or it will see this state already set, and block (rather than busy-waiting) until the state changes. The new macros effectively add another meaning to the **waiting** state of the multi-threaded signal delivery macros — which is simply that another thread is delivering a signal, and the current thread must wait before doing so.

This optimization was not necessary when signal delivery had to be performed on the master processor, since waiting to run on the master would cause a thread to block in an equivalent manner.

4.2 Per-thread suspension of execution

We have also introduced a new per-thread saved signal mask, among other state, in order to permit *sigsuspend* to operate correctly on single threads within a multi-threaded task. In a similar vein, we have ensured that only one thread can successfully *wait* for a given process to terminate or stop, and that all other threads waiting for a given process (or the last of a set of processes) will return an error indication, rather than waiting fruitlessly. (This is also explained further in Section 6.2.)

These two issues (correct per-thread suspension of execution and safe multi-threaded execution of process management calls) were the only ones related to multi-threaded tasks that we felt obliged to address (because the existing implementation was incorrect). Given the rapid evolution of proposed standards in this area, we felt it best to exclude from the project other changes to aspects of multi-threaded execution. The most obvious of these would have been adding more per-thread signal state, or augmenting signal delivery semantics.

5 Changes to Provide Multiprocessor Synchronization

We made two sorts of changes in this area. To begin with, we added locking, a reference count, or both to the various data structures associated with a process. We also introduced several global locks in order to synchronize access to kernel tables or other global structures.

The global locks we added are listed in Table 1. All but the last of these do the obvious in co-ordinating access to the table with which they are associated, and will not be further described here. Of these locks, only the `pgrphash_lock` exists in uniprocessor configurations.

Lock Name	*Lock Type*	*Associated Data*
`pgrphash_lock`	R/W	process group hash table
`uidhash_lock`	simple	process uid hash table
`pid_lock`	simple	table of per-process "handles"
`proc_relation_lock`	R/W	all process relation pointers

Table 1: *New global locks*

(Note that the lock `pid_lock` is associated with a replacement for the traditional `proc` table, in which only a minimal per-process "handle" is statically allocated, each `struct proc` being allocated dynamically on demand. This scalability enhancement has nothing to do with the parallelization project, so will not be discussed further.)

The `proc_relation_lock` is more interesting. It protects the process pointers (parent, child, and sibling pointers) in *all* active processes. Though it may seem counterintuitive that a single, global lock should be needed for this purpose, we will present a case for it a bit later. First, in part to motivate that discussion, we will look at the per-`struct` changes we made to provide MP synchronization.

Changes to three structures were central to our parallelization effort. These are the process group (`pgrp`), process (`proc`), and session (`session`) structures. The changes to each are described in the sections that follow. Of the synchronization mechanisms listed, only the process group lock `pg_lock` exists in uniprocessor configurations.

5.1 Changes to process groups

As part of our work, each process group, or `pgrp`, now contains a reference count, `pg_refcnt`, to record references to the group other than from the processes in its own member list. Such references occur, for instance, from the TTY code, when a job-control signal must be sent to a group not currently in the foreground. The `pg_refcnt` member is protected by its own simple lock, `pg_refcnt_lock`, in order to ensure that a valid reference can be made to the group through one of its member processes (see Section 7.2.1).

A `pgrp` now also contains a read/write lock, `pg_lock`, taken for writing when the `pgrp` itself or its member list is modified, and for reading whenever the `pgrp` or its member list is otherwise accessed. This means, in particular, that the lock is held across calls to `pgsignal`. Since the new implementation of `psignal` can block, we initially wanted to avoid this prolonged lock retention. However, we found that strictly obeying a lock ordering scheme could eliminate the risk of deadlock in this situation (see Section 5.5).

Further, we found that our would-be replacement for the use of this lock was not robust. We had considered replacing it with a sequence of locking operations on the `proc`'s in the member list. However, since the member list is traversed using members of the `proc`'s themselves, and since (in

this scheme) a process could change groups at any time, use of the `proc`'s alone to control group traversal could result in partial traversal of n different groups by a single `pgsignal`. The idea of fixing this scheme by locking the original `pgrp` against process removal led us squarely back to the read/write lock that we have implemented.

5.2 Changes to processes

The "process", or `proc` struct posed different problems. Numerous parts of the kernel can refer to an arbitrary `proc`. In addition, the subsystem we concentrated on is filled with references to the current process. We felt we needed different, low-cost synchronization mechanisms for these two cases, for which a blocking lock was clearly inappropriate.

To ensure that a process remained valid throughout an arbitrary reference to it, we implemented a per-`proc` reference count, used whenever a process is successfully looked up by another process. A process can begin an *exit* (and terminate all threads other than the exiting one) while still referenced — it will not complete the *exit* until all references to it have been ended.

To safeguard modifications to the contents of a `proc` struct, we also added a per-`proc` simple lock, `p_lock`, which is taken only long enough to alter the members affected in a given operation.

5.3 Change to sessions

The only change we found necessary to the `session` struct was the addition of a simple lock, `s_lock`, to control access to its contents.

5.4 Global process relation lock

As noted above, the global `proc_relation_lock` protects the four pointers in each `proc` structure to other, related processes. These are are listed in Table 2:

Member Name	*Process Designated*
`p_pptr`	parent
`p_cptr`	youngest child
`p_ysptr`	next younger sibling
`p_osptr`	next older sibling

Table 2: *Per-*`proc` *process relation pointers*

We introduced this lock in order to avoid the formidable lock-ordering problems we foresaw in trying to use per-`proc` locks alone to control changes in process relations. As an example of these, consider the following points:

- When a process *fork*s, its `p_cptr` is set to point to the new child. If the child exits before the parent does, then the parent's `p_cptr` may need to be changed to point to an older sibling (if it still pointed to the exiting child).
- As a result of a *fork*, the `p_pptr` of the child process is set to point to the parent. If the parent exits before the child does, the child must be "reparented" – that is, its `p_pptr` must be changed to point to the *init* process.

- At several points in the kernel, a child process must be able to refer to its parent reliably (to send it a `SIGCHLD`, for instance, if the child stops due to a signal or because it is being traced). Similarly, a parent process must be able to refer reliably to its youngest child, in order to begin a traversal of the list of parent's children (which is maintained via the sibling pointers of each each child).
- We provide a reference count, of course, which is incremented to prevent a process from exiting while still in use by another process. However, in order to use the count in the contexts outlined here, we need to lock two processes at once, in an order determined by the operation being performed (such that we can't order the acquisition of the locks by any fixed scheme).

Thus in the absence of the `proc_relation_lock`, attempting to lock pairs of processes in the orders required could (on an MP system) easily result in deadlock. Placing all changes to these process pointers under the control of a single, global lock, however, permits us to access the pointers within related processes, and increment the reference counts of the target processes, without individual locks being taken beforehand.

5.5 Lock Ordering

More generally, in order to acquire multiple locks concurrently with no risk of deadlock, we defined a fixed order in which different kinds of locks had to be taken. A partial listing of this lock ordering (containing only the locks mentioned in this paper) follows, in Table 3. In general, blocking locks must be acquired before simple locks, and, subject to this constraint, global locks must be acquired before local locks.

Order	*Lock*	*Type*	*Scope*	*Comments*
1	`pgrphash_lock`	r/w	global	
2	`proc_relation_lock`	r/w	global	
3	`pg_lock`	r/w	per-`pgrp`	If two needed, order by address.
4	`uidhash_lock`	simple	global	
5	`pid_lock`	simple	global	
6	`p_lock`	simple	per-`proc`	If two needed, order by address.
7	`pg_ref_lock`	simple	per-`pgrp`	
8	`s_lock`	simple	per-`session`	

Table 3: *New lock Ordering*

6 Parallelized Process Management

Enabling the process management system calls to run on any processor in an MP system introduced two sorts of problems. The first concerned locking data written at high spl. The second was a variety of races that then became possible, both between related processes and between threads in a multi-threaded process.

6.1 Accessing data written at high spl

In an OSF/1.0 kernel, two pieces of process state can be written from the clock interrupt processing code, which runs with most (usually all) interrupts disabled. The first datum is a flag, used only

if user profiling is in effect, indicating that the profiling buffer should be updated on behalf of the thread interrupted by the clock tick. The second datum is the current CPU time limit for the current process. If this limit is exceeded, a `SIGXCPU` signal is sent to the process, and the limit is increased, to allow the process time to process the signal.

Both of these data must be protected from simultaneous writes by different processors. In the first case, we moved the "profiling buffer update" flag into its own word in a per-thread struct, where only the current thread would ever access it. In the second case, we used the existing per-process `u_time_lock`, which already had to be acquired with interrupts disabled, to control access to the CPU limit.

Once we introduce an operation on a lock in interrupt context, of course, we must ensure that the lock involved is always acquired at high spl. In thread context, this means disabling any interrupts during which the lock may be acquired. To see why, suppose a thread acquiring a lock that is also acquired in interrupt context fails to disable that interrupt. If the interrupt occurs and the interrupt handler tries to acquire the lock, the handler will spin waiting to obtain it. But the handler will spin forever in this case, since it interrupted the thread holding the lock (see [10] for a full discussion of this and other MP synchronization issues).

6.2 Resolving process management races

A number of races could be avoided simply by careful ordering of state changes. This was particularly relevant in contexts where such ordering had not mattered, when the code executed only on the master processor.

One such race involved the code implementing *wait*-style system calls (which enable a parent process to await or simply check for a state change in one of its children) and the code in *exit* that actually terminated a child. The child had to ensure that its termination was complete before awakening its parent, since otherwise the parent might run (and fail to see that the child had exited) before the child could complete its processing.

A more interesting race in `wait` would occur if multiple threads in a parent process each waited for the same child (or overlapping sets of children). To ensure that each parent thread would either wait for a child successfully or (correctly) return an error indication, we made the following changes in the *wait* code:

- Before traversing the child list of the calling process, take the `proc_relation_lock` for reading.
- If a child is found and is in the desired state, set a flag indicating that the current thread "owns" the child, unless of course the flag is already set.
- If a child is found, but is already owned by another thread, forget that we found it and continue scanning the child list.
- Once we have flagged the child as ours, release the `proc_relation_lock`.
- If we're removing the flagged child from its sibling list (i.e., if the child has terminated), re-acquire the `proc_relation_lock` for writing and update the list.

A further race condition existed between the *fork* system call and *exit* or *exec* when the two were used concurrently by different threads in the same process. The utility of such a combination is questionable, at best, but we did want to ensure that all kernel state would remain consistent if the combination occurred. This we did simply by adding a `p_forkcnt` member to the `proc` struct, to count *fork* operations currently in flight. Before a *fork* is allowed to proceed, the `sig_lock` is acquired, ensuring that no other thread in the process has begun an *exec* or an *exit* (see Section 4.1). Then `p_forkcnt` is incremented.

In *exec* or *exit*, on the other hand, a non-zero fork count causes the system call to block, waiting for the count to return to zero. At the end of a *fork*, the forking thread in the parent process decrements the count and awakens any blocked threads.

7 Parallelized Signal Handling

7.1 Reliable Signal Generation

One problem area in signal process is the signal generation code — `psignal`, which signals a single process, or `pgsignal`, which signals each process in a `pgrp`. In an ordinary uniprocessor kernel, this code does not acquire any locks, so does not block, and may be run at either interrupt or base level. On an MP, however, both `pgsignal` and `psignal` must acquire locks. Since the locks involved are widely used at base spl, they cannot be used at high spl as well.

Thus for configurations in which the signal generation code can block, we defined a deferred signalling mechanism, which we refined and extended from one used for an unrelated purpose in OSF/1.0. In this mechanism, a signal generation request enqueues an event containing the signal parameters, for later processing by a dedicated kernel thread. Naturally, the kernel thread runs at base spl, eliminating locking problems.

To simplify the correct use of `psignal` and `pgsignal`, we have defined the symbols as macros, which, in each kernel configuration, expand to whichever set of routines (direct or indirect) is universally safe in that configuration. Thus these macros may always be used. If a developer knows that a given call will always occur at base spl, he may explicitly call the direct version of the routines, rather than using the macros.

Function (or Macro) Name	*Usable From*	*Operation Type*	*Target*
`psignal` (macro)	any context	configuration-dependent	process
`pgsignal` (macro)	any context	configuration-dependent	process group
`pgsignal_self` (macro)	any context	configuration-dependent	process group + self
`psignal_inthread`	thread context	immediate	process
`pgsignal_inthread`	thread context	immediate	process group
`pgsignal_inthread_self`	thread context	immediate	process group + self
`psignal_indirect`	any context	deferred	process
`pgsignal_indirect`	any context	deferred	process group
`pgsignal_internal`	internal only	immediate	selected via flag

Table 4: *Signal generation macros and entry points*

Table 4 presents a summary of the signal generation macros, and of the underlying routines. Each macro expands to one of the corresponding routine names, according to kernel configuration. ("Thread context," of course, means base spl level. The symbols `pgsignal_self` and `pgsignal_inthread_self` are explained in Section 7.2.3.)

7.2 Process group signalling

Numerous problems arise from the fact that, in the parallelized process management code, one thread in the current process may change the `pgrp` of the process, while another thread is attempting to signal the `pgrp` (or otherwise traverse its member list).

7.2.1 Referring to a pgrp reliably

In order to signal a pgrp, the first problem a process must overcome is just to refer to the pgrp reliably. Generally, a pgrp is accessed via one of its member processes (i.e., using the p_pgrp member of a proc). To keep the process from changing groups while its p_pgrp pointer is read, the process is locked "around" the operation. As soon as the lock is released, the process can change groups. Further, if its former pgrp then becomes empty, the pgrp itself can be deleted. If the thread that read the pointer to the pgrp were interrupted for sufficiently long, both of these events might occur before it even got around to using the pgrp pointer it had acquired.

Clearly, to ensure that the pgrp remains valid, a reference should be taken on it while holding the lock of the process through which we accessed it. The reference count, however, must be incremented under lock. But taking a pgrp pg_lock while holding a proc lock would violate our locking hierarchy, introducing potential deadlocks. So we added the per-pgrp pg_ref_lock, at the bottom of the hierarchy, whose only role is to control access to the group reference count. This lock, which can be taken while the current proc is still locked, ensures that the reference count can be reliably incremented, and hence that the pgrp will remain valid after the process is unlocked.

This lock does not address the possibility that the process will change groups, once it is unlocked. We discuss that now.

7.2.2 Indeterminacy of pgrp signalled

One apparent problem that is actually not an issue in most respects is the fact that the current process may no longer belong to its original pgrp by the time the pgrp is signalled. Certainly it would be possible in some way to prevent the process from changing pgrp's while preparing to signal its (once-current) pgrp. (A process is already prevented from changing groups while group signalling is in progress — see Section 5.1.) However, the group change could always then occur as soon as the process was released — in particular, before it or any other group member had had time to act on the signal.

In general, therefore, it serves no particular purpose to delay a change of pgrp while deciding whether to generate a group signal.

7.2.3 Signalling the current process

Though, as just noted, the current pgrp of the current process is not germane when signalling the (potentially not-current) pgrp recorded for it at some previous point, we do often need to ensure that we signal the current process, whether or not we wind up signalling the pgrp to which it belongs at the time of the signalling operation. This need arises in signal generation performed by the TTY code, for instance. This poses a problem only in a multi-threaded process, in which one thread changes the process' pgrp while another thread is preparing to signal the previous pgrp.

To deal with this situation, we created the new macro pgsignal_self, and the new underlying routine pgsignal_inthread_self. These are like their ordinary pgsignal equivalents, except that they ensure the current process is always signalled, even if it is not a member of the pgrp indicated.

Since this behavior does differ from that of ordinary group signalling, of course, the entry point must be kept separate from pgsignal.

8 Observed Parallelization Gains

The parallelization changes we made to the OSF/1.1 kernel, in conjunction with other changes not described in this paper, have eliminated the use of unix_master in the course of normal kernel

usage in a BSD environment. There remain subsystems (primarily System V IPC and the System V filesystem) that are unparallelized, but, outside these, very few uses of `unix_master` remain anywhere in the kernel.

We believe that reducing the serialization induced by `unix_master` has yielded a significant increase in system-wide throughput, and has produced a measurable gain in single-threaded throughput for applications heavily dependent on the parallelized system calls. Note, in particular, that an interactive shell (whether or not a job-control shell) will issue several signal-related system calls each time it executes a command line. Now, these no longer force context switches in order to run on the master processor.

To quantify the gain in performance introduced by our changes, we will present comparisons of a kernel not containing our changes with the current OSF/1.1 kernel. These comparisons provide only a general impression of the performance enhancements involved, since the two kernels we used differ by more than just our changes. We have tried to minimize the impact of unrelated changes on the comparisons, but we will point out the results we consider especially affected by them.

The machine configuration used for these tests was an Encore Multimax containing 224 MB of memory and ten NS32332 processors, each rated at about 1.5 MIPS. All tests made use of a single SCSI disk attached via an EMC interface. The first three tests were run on an idle machine in single-user mode, while the pseudo-user session test was run on an idle machine that was up multi-user, with its network interface enabled. All times given below are elapsed (real) times, as measured either with the free-running microsecond counter provided by the Multimax (in the first two cases), or with `/bin/time` (in the last two cases).

8.1 Signal delivery throughput

We begin with a statistic very closely tied to our changes — namely the number of signals that can be delivered to a process per second. Figure 1 shows the delivery throughput in the two kernels, with from one to ten process pairs concurrently sending and receiving `SIGUSR1`.

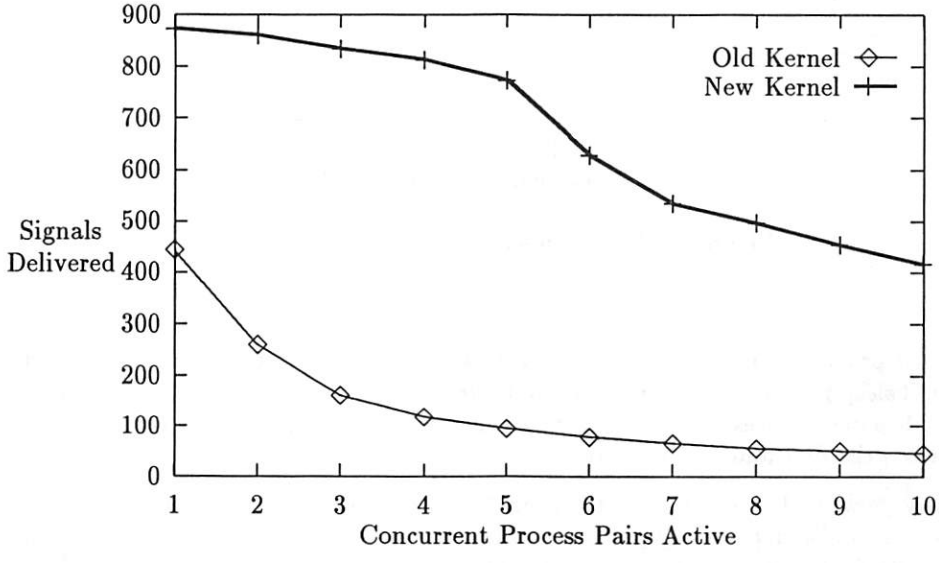


Figure 1: *Signal Deliveries per Process per Second*

In each pair of processes, a receiver process set up a handler for the signal, then looped over

calling *sigsuspend* in order to await signal receipt. The other process looped over sending the signal to the receiver using *kill*. When 1000 signals were received, the receiver exited, and the sender, when *kill* failed, did likewise. We measured the elapsed time in the receiver between the first call to *sigsuspend* and the final signal receipt.

In the old kernel, throughput per receiver process varies inversely to the number of active receivers, as would be expected. In the new kernel, the dropoff in performance with more than five sender/receiver pairs active is probably due to context switch overhead, since naturally the senders and receivers were running simultaneously and only ten processors were present.

8.2 Fork completion throughput

Another statistic of interest, somewhat more difficult to measure, is the number of forks that a process can complete per second. Figure 2 shows fork throughput in the two kernels, with from one to seven processes simultaneously calling *fork*.

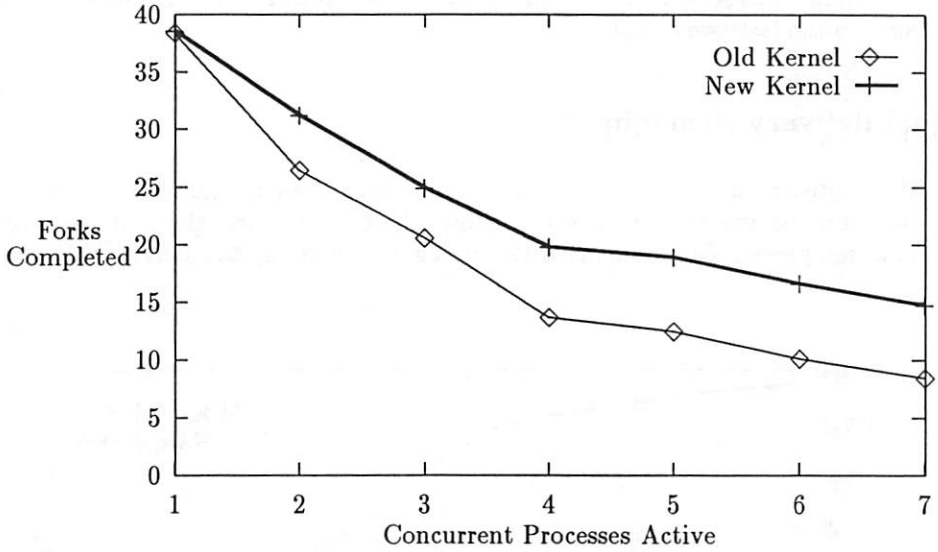


Figure 2: *Forks Completed per Process per Second*

Here, each process would loop over calling *fork* fifty times, as quickly as possible. Each child started would sleep for long enough to allow all forks to complete, then would exit. After completing all forks, each parent process would wait for its children to exit. We measured elapsed time in the parent between the first and the last fork.

Obviously, *fork* involves much more than just the process manipulation code that we parallelized — in particular, it also depends heavily on the Mach VM subsystem in creating child address spaces. Thus our changes, though still having a noticeable effect on throughput, did not produce the same magnitude of improvement as was possible in signal handling. These are the numbers that we consider the least useful for evaluating our changes, since the VM subsystems differ between the two kernels used, and since this test centers on an operation that is VM-intensive. At the very least, the single-process numbers do show that our MP locking does not degrade performance in this case.

8.3 Concurrent build throughput

To get some performance data more closely related to real-life system usage, we also measured the elapsed time needed to complete from one to eight instances of a medium-sized *make* operation, all executing simultaneously. Figure 3 plots the average elapsed time per build against the number of instances running.

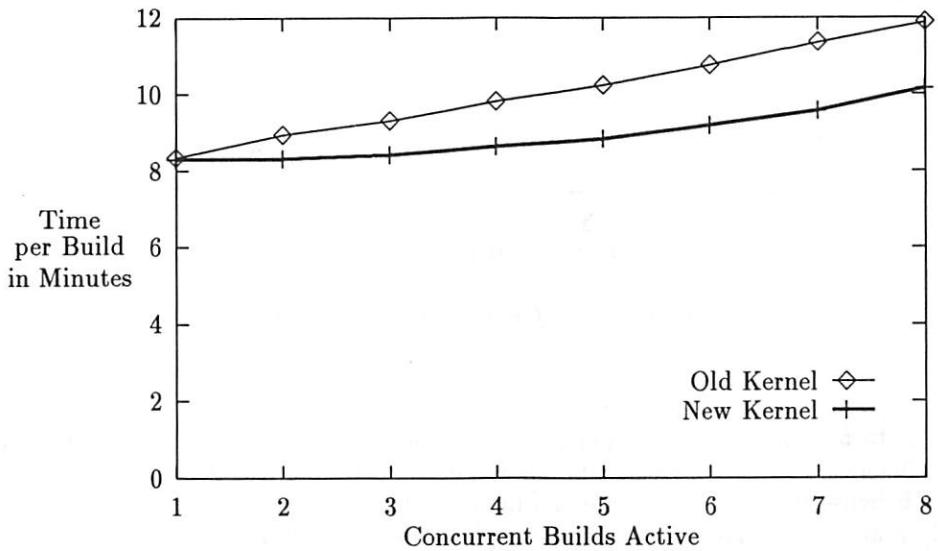


Figure 3: *Elapsed Time for Multiple Builds*

Each build was single-threaded, and compiled separate copies of the same source files, which contained roughly 10000 lines of C code. The builds were performed in sibling subdirectories of the same parent directory.

With the new kernel, it is likely that the sharper performance degradation seen to the right of the graph results more from loading on the single disk channel in use than from kernel data contention. Naturally, in this case even more than in our *fork* measurements, the improvement introduced by our changes is dominated by the time required for operations unrelated to process management.

8.4 Pseudo-user session throughput

The final data we will present here result from runs of a special script designed (very loosely) to emulate the behavior of a logged-in user, by exercising a broad subset of the OSF/1 utility set. The script is used by defining a number of pseudo-users, and giving them entries in `/etc/passwd` that specify the script as their "login shell." A remote front-end command then logs in to the test machine, specifying one of the pseudo-users as its login id. A run of this script subjects the system to significantly higher load than a real user would, since there are no delays between commands executed.

Figure 4 plots the elapsed time needed for the execution of one iteration of the pseudo-user script against the number of instances of the script being run simultaneously.

By the nature of the test, the numbers here are less well-ordered than any of the others we have presented. Though the test system was otherwise idle, it was up in multi-user mode and accessible to a local network, as we noted above. Also, in this particular case, we arranged for each concurrent

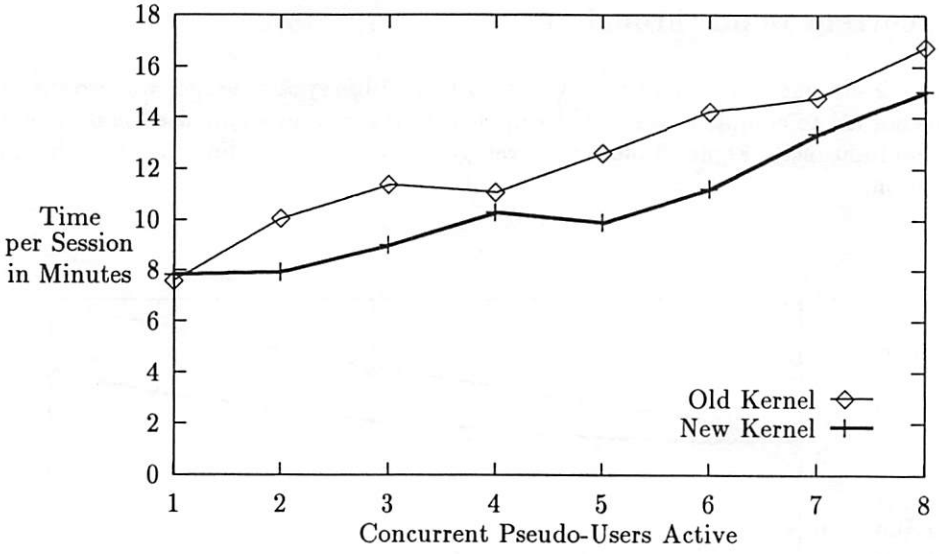


Figure 4: *Elapsed Time for Multiple Pseudo-User Sessions*

session to be started after a delay dependent on its order in the set of sessions currently being run. Thus different numbers of concurrent sessions could interact in unexpected ways, counteracting the uniform increase in per-session elapsed time one might otherwise expect to see. Really, the primary information to be gleaned from this graph is that the new kernel will, in general, execute this multi-user workload faster than the old kernel would.

9 Conclusions and Future Directions

We have successfully introduced significant performance gains into the OSF/1 kernel by parallelizing its process management and signal processing subsystems. As the previous section shows, the new kernel containing our changes in general degrades more gracefully under load than does the older kernel we compared it against. In the only test for which this was not true (the pseudo-user session test), though the shape of the performance curve is similar, our kernel's performance is still consistently better by a noticeable factor.

We believe that our implementation strikes a reasonable balance between efficiency and the complexity of the changes introduced. There is, of course, room for improvement in what we did. The most noticeable bottleneck that could be removed is the global `proc_relation_lock`, though experience with equivalent locks in other parallel kernels in the past has not shown them to be a major problem[3, 12].

We also have some data on contention for this lock in our kernel, under moderately heavy load (namely the eight-pseudo user session test running concurrently with the eight-process build test). When acquiring a lock, a thread can be obliged to wait in two different ways. First, when a lock cannot be immediately acquired, the locking code will busy-wait for a short time before blocking the calling thread. Then, if the lock remains inaccessible at the end of that time, the caller is blocked. We found that a thread had to wait at all in acquiring the `proc_relation_lock` well under two percent of the time. Of these waits, ten percent involved blocking the calling thread — meaning that 90 percent of the waits were of very short duration.

This contention rate, though above the median for locks used elsewhere in the kernel, does not make the `proc_relation_lock` one of the most heavily-contended. However, the lock might pose a scalability problem in much larger systems. It is possible that further per-`proc` locks could be used to replace the global lock, though more thought is still needed to be sure that these would be equivalent. One could imagine adding:

- a simple lock to control access to the process reference count alone, which (like the equivalent `pg_ref_lock`) would be at the bottom of the lock hierarchy, and hence could be taken without conflicting with the use of general per-process locks (`p_lock`).
- a lock, probably a simple lock, to control access to the child list of a process, that is, to the sibling pointers of the process' children.

With these in place, one could more easily synchronize changes to inter-process relationships "naturally" (i.e., by following the semantics of the code, rather than the constraints of a lock hierarchy). However, correctness concerns aside, we would be reluctant to add still more per-struct locks to our implementation without compelling evidence they were needed. Though performance measurement is still underway, we don't feel at the moment that we have such evidence.

Further analysis, particularly using lock statistics, may well also reveal other opportunities for optimization.

In conceptual terms, this project resolved some of the conflicts between traditional Unix abstractions (processes and signals in particular) and the multi-threaded execution model provided by Mach and hence by OSF/1, and began to explore the provision of a consistent and usable interface between the two. Much remains to be done, especially in promoting threads to be full-fledged participants in the Unix scheme. We hope we have laid some groundwork for future efforts, as the relevant standards mature sufficiently to become useful in guiding them.

10 Acknowledgements

We would like to thank the many persons at OSF and at Encore who supported us in the design and implementation of this project. David Black, of the OSF Research Institute, implemented the signal processing macros and other extensions for multi-threaded processes present in OSF/1.0, which served as a basis for our own changes to multi-threaded process manipulation. He also provided valuable advice and commentary throughout the project, as did several persons in the OSF engineering organization, including Jeff Carter, Jeff Collins, George Feinberg, Gary Fernandez, David Mitchell, and Tom Talpey. Finally, Alan Langerman, our project leader at Encore, provided helpful insights when we most needed them. His extensive SMP expertise significantly eased project development.

References

[1] M. Bach. *The Design of the Unix Operating System*, Englewood Cliffs, NJ, 1986. Prentice-Hall, Inc.

[2] J. Boykin and A. Langerman. The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis, in *Conference Proceedings, USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, pp. 105-126, Ft. Lauderdate, FL, 1989. USENIX Association.

[3] Personal communication from Jeff Collins, who designed the parallelized process management subsystem in Encore UMAX 4.2.

[4] F. Douglis. Experience with Process Migration in Sprite, in *Conference Proceedings, USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, pp. 59-72, Ft. Lauderdate, FL, 1989. USENIX Association.

[5] D. Freedman. Experience Building a Process Migration Subsystem for Unix, in *Conference Proceedings, 1991 Winter USENIX Technical Conference*, pp. 349-356, Berkeley, CA, 1991. USENIX Association.

[6] G. Hamilton and D. Code. An Experimental Symmetric Multiprocessor Ultrix Kernel, in *Conference Proceedings, 1988 Winter USENIX Technical Conference*, Berkeley, CA, 1988. USENIX Association.

[7] S. Leffler, M. McKusick, M. Karels and J. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*, Reading, MA, 1989. Addison-Wesley Publishing Company.

[8] J. Lions. *A Commentary on the Unix Operating System*, Sydney, Australia, 1977. Univ. of New South Wales.

[9] S. LoVerso, N. Paciorek, A. Langerman and G. Feinberg. The OSF/1 Unix Filesystem (UFS), in *Conference Proceedings, 1991 Winter USENIX Technical Conference*, pp. 207-218, Berkeley, CA, 1991. USENIX Association.

[10] N. Paciorek, S. LoVerso and A. Langerman. Debugging Multiprocessor Operating System Kernels, in *Symposium Proceedings, Symposium on Experiences with Distributed and Multiprocessor Systems*, pp. 185-202, Atlanta, GA, 1991. USENIX Association.

[11] U. Sinkewicz. A Strategy for SMP ULTRIX, in *Conference Proceedings, 1988 Summer USENIX Technical Conference*, pp. 203-212, El Toro, CA, 1988. USENIX Association.

[12] *UMAX 4.2 Programmer's Reference Manual*, Marlborough, MA, 1986. Encore Computer Corporation.

Mach Resource Control in OSF/1

David W. Mitchell
October, 1991

Abstract

All systems have inherent restrictions imposed by their hardware architecture and configuration; to ensure reasonable operation, these limitations must not be exceeded. Timesharing systems may also need to impose constraints on the consumption of an individual user to ensure that all users can get adequate access to system services. Systems layered upon Mach require a considerable amount of extra code to impose consistent and secure limits if Mach system services are also exported to users.

This paper describes the design and implementation of a general framework for controlling Mach resources which permits servers or other layers of kernel code to account for and control resource consumption in the Mach layer.

1. Introduction

All systems have inherent restrictions imposed by their hardware architecture and configuration. These include such things as support for only a finite amount of address space or physical memory, the ability to run a fixed number of processes, or to handle only a certain number of pending I/O requests. To ensure reasonable operation, these limitations must not be exceeded. At a minimum, the constraints imposed by hardware must be enforced on processes running on the system [1]. It is often desirable to further confine the consumption of any individual process in order to prevent one process from interfering with another. Commercial timesharing systems frequently provide mechanism for controlling the resources used by any single user to ensure that each user of the system can obtain access to the resources needed to make progress. Though workstation operating systems typically treat all users equally, in the commercial world users with more money get more resources. Some mechanism for controlling resource usage is necessary to make this feasible.

2. The Structure of OSF/1

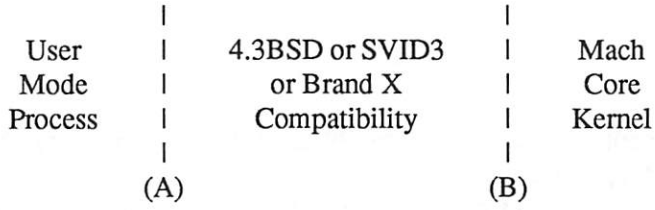
The OSF/1 operating system is an example of a layered operating system, consisting of two basic layers of kernel code. The core is derived from the Mach (2.5) operating system, which supports a small number of fundamental abstractions implemented by a communication oriented kernel which supports multiprocessing [2]. The basic abstractions exported are the task, thread, port, message, and VM object.

- A **task** comprises the execution environment in which threads may run, and is the basic unit of resource allocation, consisting of a virtual address space and access to various resources via ports.
- A **thread** encapsulates the state of a single run-time execution path within the task.

- A **message** is a typed collection of data objects, used for communication between threads. A message may be of any size, and is sent over a port.
- A **port** is a communication channel which is logically a queue of messages protected by the kernel.
- A **memory object** is a collection of data managed by a server which can be mapped into the address space of a task.

The outer layer of OSF/1 provides 4.3BSD and SVID3 functionality and has also been parallelized to enhance multiprocessing support [3].

Currently, these two layers are bound together and both run in supervisor mode on each hardware base. Ongoing research [4] is investigating the efficacy of restructuring these two layers so that only the Mach core ("micro-kernel") requires privileged mode. The 4.3BSD code and other compatibility layers would then run in user mode in one or more server processes.



Today, boundary 'A' is a "real" boundary in the sense that it is enforced by the memory management hardware, and boundary 'B' is one of software convention only. In the future, boundary 'B' may become the one enforced by hardware, with 'A' becoming simply the inter-process boundary between one or more server processes and other user-mode processes.

3. Motivation & Goals

The Mach core of OSF/1.0 has no notion of resource limiting, other than allocations failing due to complete exhaustion, though the Mach Kernel Interface Manual [5] lists resource limits among the capabilities contained within tasks. The compatibility layer partially enforces the BSD notion of limits, allowing a process control over:

- cpu time used
- maximum size of a file that can be created
- maximum size of the data segment (break value)
- maximum size of the stack
- maximum size of a core file
- maximum amount of physical memory used (resident set size)

Unfortunately, the traditional model of text/data/stack does not map well onto the more general virtual memory model of OSF/1, and in any case the data and stack limits are easily circumvented by direct use of the Mach primitives. Enforcement of limits on entities whose underlying implementation consumes system memory, such as tasks, threads, and virtual space is

nonexistent. An errant or malicious process may easily cause the kernel to allocate a large percentage of physical memory for internal use, leaving too little for user processes to run efficiently. There is a clear need for a general framework, usable by all subsystems, which will support accounting of, and limits to, the consumption of arbitrary Mach resources. This framework must also provide other layers of code with the ability to control and monitor these resources. It would also be useful if this framework was structured in such a way that it could be migrated easily into the micro-kernel research effort.

While there is a need to limit such potentially exhaustible system entities as:

- tasks
- virtual space
- threads
- ports, port sets, port names
- processor sets
- address map entries

on a per-user, per-session, or some other basis, it is also clear that this enforcement is most appropriately, efficiently, and securely done by code in the Mach core, though this layer has no knowledge of processes, users, or sessions. Hence, the approach taken here is to construct a general mechanism in the Mach core of OSF/1 which permits secure accounting of resource consumption on the basis of a task or arbitrary group of tasks with sufficient hooks and functionality to allow the outer layers of kernel code to control resource usage among groups of tasks on the basis of whatever abstractions that layer implements. This code was also designed so that it could be easily reusable in pure Mach, distributed environments and the micro-kernel. The interfaces exported from the Mach layer will be exported via MIG [6].

4. Structure & Function

To solve the problem of accounting for, and constraining resource consumption in a task, it seems natural to add an object which will encapsulate the required bookkeeping. In financial accounting, a ledger is a book containing accounts, to which debits (debt items) and credits (asset items) are posted from the original transaction records. Hence the metaphor for this new accounting object is that of a ledger, in which the transactions of the task's threads are posted in the form of debits and credits. For each Mach resource to be accounted for, there is a separate line item containing a balance and maximum. The balance tracks the amount of the resource consumed. There is a function which will debit the ledgers to account for resource consumption, and another to credit the ledgers when the resources are returned to the system pool. Each ledger has a maximum, setting a firm limit past which the ledger may not be debited. Those with backgrounds in accounting, noting the small loss of generality compared to an actual bookkeeper's ledger, may think of this new ledger object as a task's resource account with the system, containing an implicit right hand side of zero.

Similar functionality could have been obtained by having only a balance, which gets initialized to the maximum usage and is adjusted as resources are allocated and deallocated, never being allowed to go below zero. However, this would not permit the maximum to be adjusted once the ledger was in use without requiring implicit knowledge of the ledger's initial state.

A task has two ledgers attached to it: a private ledger which accounts for the resource usage of an individual task, and a shared ledger, which accounts for resources used among a group of tasks. All line items exist in both ledgers, and the routines which debit and credit items always modify both ledgers. Having two ledgers, one used per-task and another allowing a set of tasks to be controlled provides flexibility in implementing useful resource control on top of this facility. The per-task ledger is intended to provide constraints, which may be modified by the task. No privileged capability is required to access the per-task ledger, providing discretionary controls which are also useful in a pure Mach environment, without the intervening compatibility layers. The shared ledger is intended to enforce mandatory constraints set by the system administrator, and so requires that a certain capability (privileged host port) be held in order to modify the maxima.

Having a private per-task ledger also makes it possible to correctly account for a single process's consumption when it is necessary to detach the process from one shared ledger and attach to another. Since the private ledger details the resource usage accountable to each task, the change of shared ledgers is straightforward. Moving the task to a different shared ledger is necessary when the BSD code layer changes the process' real uid, such as during login.

Note that some maxima may be set to "don't care" or "infinite"; for example, limiting virtual space per user is not generally useful, and limiting tasks per task is inane.

This structure provides separation of mechanism and policy. While the Mach layer provides the hooks to account for threads, tasks, virtual space, etc., the BSD layer or server(s) may use the shared ledgers to implement resource limits on a per-user, per-session, or any other useful basis. Also, the per-task ledger may be set differently for each task. Providing for a list of ledgers was considered, but the increased generality was ruled out in favor of efficiency. Multiple shared ledgers gave little added functionality, since the outer layers of code can control the domain over which the sharing takes place and these layers are free to set the shared maxima on any basis desired, such as the lowest or highest usage allowed to any group of which the task is a member.

5. Implementation

A ledger is a simple data structure, containing a reference count, a lock, some number of line items each containing a balance and maximum, and a count of the number of line items.

Task_create() sets up each task with a private ledger of its own, with maxima inherited from its parent's private ledger. A task's shared ledger is inherited from its parent. If there is no parent, new ledgers are used, with maxima set to LEDGER_UNLIMITED.

Subsystems participating in resource control use the debit and credit functions to check whether further allocation is allowed, and to account for resources returned to the system, respectively.

The VM subsystem is one notable exception, in that it does not store the maximum and balance in the ledger structure, but forwards the data into the task's address map, for easy access by the lowest level VM routines. Therefore the VM subsystem does not use the debit and credit functions but maintains the balance as it always has, now validating against the maximum before allowing a change in size. The ledger code references the values in the task's address map when reading or writing the maximum or reading the balance.

The compatibility layer of OSF/1 currently uses the per-task ledger to implement discretionary controls such as the per-process address space limit of SVID3. The shared ledger is used to enforce mandatory limits such as the limit on processes per real uid mandated by POSIX 1003.1. This limit cannot be overridden by the user.

In the BSD layer, the first process has its ledger maxima set to the system defaults. When a new process is started, it inherits its private ledger maxima from its parent and uses the same shared ledger as its parent. Since only descendants of init will have their maxima set, any Mach system tasks will have unlimited access. These system defaults are included among the parameters which can be set by the system administrator at system boot time.

When a process changes its real uid, as happens during setuid(), it is necessary to change the shared ledger attached to the underlying task. The OSF/1 internal set_uids() routine, if it is changing the real uid, sets a flag in the process which is checked when a process tries to exec. During exec, set_per_user_limits() is called to attach the process to the shared ledger of another task belonging to the same user, if such a task exists. If there is none, set_per_user_limits() procures a new shared ledger to be used by all processes of that user. This is done by detaching the process from the default (system-wide, unlimited) shared ledger, attaching to the new one, and setting the maxima appropriately for the process's privilege. Processes belonging to superuser, or in security configurations, processes possessing the SEC_LIMIT privilege, have their maxima left unlimited. If an existing process belonging to that user is found, its shared ledger is attached to the new process provided that no system limit is exceeded, otherwise an error is returned. Calls that set only the effective uid, (e.g.: execve() and exec_load_loader()) do not need to do this, since accounting is attributed to the actual user, as indicated by the real uid.

This baroque structure was chosen to avoid adding a new failure scenario to the setuid(2) call, which would have had the nasty implication that a process might unwittingly be left running with high privilege after an attempt to lower its privilege. In practice, delaying the accounting check until the next exec call after the setuid operation does not allow a loophole, since setuid(2) is typically a one-way door out of privileged mode. For example, login would be unable to exec a shell and would fail. A user who attempts to exceed a limit by running a privileged program finds that another task cannot be started in which to run that program, since the process is accounted against the real uid.

Care was taken to layer the functions needed to implement this new facility, so that they could be moved easily into the micro-kernel environment. The routines which manipulate ledgers can be logically separated into three groups:

- internal to the Mach layer

ledger_create()	used by task_create
ledger_reference()	used by task_create
ledger_deallocate()	used by task_deallocate

- used by Mach subsystems participating in resource control

task_ledger_debit()	used by vm_allocate, thread_create, task_create, ...
task_ledger_credit()	used by vm_deallocate, thread_deallocate, task_deallocate, ...

- exported to BSD layer, server or other user code

task_private_ledger_read() task_private_ledger_write()	used by initialization code, setrlimit & others
task_shared_ledger_read() task_shared_ledger_write()	
task_shared_ledger_replace() task_shared_ledger_attach()	used by BSD layer to implement per-user limits

Of the routines exported out of the Mach layer, the latter four routines (which modify the shared ledger) all require the caller to supply a privileged port. Imposing this requirement prevents the process from overriding limits set up by the system administrator. The task's private limits may be changed at any time.

6. Unsolved Issues

This work provides the hooks necessary to account for, and control, the use of Mach kernel resources. It prevents an errant or malicious user from draining some key system resources and eliminates a source of covert channels. This work avoids the larger issues of how these limits can be set automatically in response to changes in configuration and load, and makes no attempt to solve the problem of providing each process or user with their fair share of system throughput under varying conditions [7].

7. Current Status

This work is part of a larger effort to commercialize OSF/1 by implementing controls on resource consumption, making fundamental system parameters modifiable either dynamically or at boot time, and eliminating system crashes due to resource allocation panics. The task_ledger facility has been used to implement the SVID3 per-task address space limit and per-user limits on tasks and threads with defaults which can be set by the system administrator at system bootstrap time. There has been no measurable performance impact. Other potentially exhaustible Mach kernel resources will soon be added into the framework.

References

1. S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman, "The Design and Implementation of the 4.3BSD Unix Operating System", Addison-Wesley, Reading, MA (1989).
2. M. J. Accetta, et al., "Mach: A New Kernel Foundation for Unix Development", Proceedings of Summer Usenix, July 1986.
3. J. Boykin and A. Langerman, "The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis", Workshop Proceedings, Usenix Workshop on Experiences with Distributed and Multiprocessor Systems, 1989.
4. D. Golub, R. Dean, A. Forin, R. Rashid, "Unix as an Application Program", Proceedings of the Summer 1990 USENIX Conference, pp. 87-96.
5. R. V. Baron, D. L. Black, et al., "Mach Kernel Interface Manual", April 1990, CMU.
6. R. P. Draves, M. B. Jones, M. R. Thompson, "MIG - the Mach Interface Generator", August 1989, CMU.
7. J. Kay, P. Lauder, "A Fair Share Scheduler", Communications of the ACM, January 1988, V.31 No.1, pp. 49-55.

Mach Interfaces to Support Guest OS Debugging

Rand A. Hoven

Hewlett Packard, Inc.
Apollo Systems Division
rand@apollo.hp.com

ABSTRACT

Many operating systems that may be implemented as servers on Mach will need to provide specialized debugging features. In general, these features can be provided just by creating the complete interface for the guest OS. However, programs operating under one of these guest OSs may desire to use features that are available only under Mach. In many cases the guest OS's debugging interface is inadequate to deal with programs using these Mach features. Furthermore, it is also desirable to be able to implement one debugger that can debug any program regardless of what guest OS the program and debugger are running under. This paper presents a mechanism by which a guest OS can present debug information to a debugger running under any guest OS.

1. Introduction

Mach is intended to be a µkernel. As such it only provides some basic features such as ports, tasks, threads, and memory objects. It is intended that any other functionality that a program might require be provided by user space servers. The program contacts the server with a request which the server fulfills. [1].

A logical conclusion from this concept is that any operating system can be emulated on top of the µkernel just by creating a server that provides all of the semantics of the desired OS. In this case the µkernel would be the host OS and the emulated operating system would be the guest OS. [2].

The debug interfaces provided by a guest OS will also be presented as part of its overall interface. This is sufficient if a program being debugged stays within the confines of the features of the guest OS. Many new programs will be written with the intention of running under a Mach guest OS and will use Mach features in addition to those traditionally provided by the guest OS. One of these features is the ability to have multiple threads in a single task. Not all guest OS debug interfaces are capable of dealing with this situation. These guest OSs will have to be extended to provide support for debugging these programs. It is also desirable to be able to provide a system with one debugger that can manipulate any program in the system regardless of the guest OS the debugger and the debugged program run under. All of this suggests that a uniform interface for debugging should be developed.

2. The Requirements

A debugger is a application that can control and modify the execution state of one or more targets. A target is a task with one or more threads executing in it. An application is a collection of tasks and threads that are cooperating to accomplish something. Currently, there are a number

of debuggers that are quite good at debugging an application that uses a single thread in a single task. While most of these debuggers often invoke the application themselves, some are able to attach to an application that has been invoked by a third party. Although the set of applications that can be debugged by these debuggers is large, it does not include some of those most difficult to debug. Some applications that cannot be handled by these debuggers include multi-threaded applications and distributed applications. [3]. "Distributed applications" refer to an application that requires more than one task to function. These tasks can be on the same or different machines. A debugger that is capable of debugging all of these applications must be able to do the following:

- Create a debugging relationship with any of the user's tasks with no changes in the task's operating environment.
- Capture the creation of a new task or thread and debug it in addition to the original target.
- Detect that new code has been loaded into the target before any of it is executed.
- Examine a task's state before it is destroyed.
- Control the delivery of asynchronous events.

In order to support the above functionality, the OS must provide the debugger with the following capabilities:

- Read and write the target task's memory.
- Read and write the target thread's registers.
- Suspend and resume execution of the target threads.
- Notify the debugger when the executable image has changed.
- Notify the debugger when a target creates new tasks.
- Notify the debugger when a target creates new threads of execution in the current task or another task.
- Notify the debugger when the last thread of a task dies, before the task is destroyed.
- Notify the debugger when asynchronous events are delivered to the target threads.
- Support a debugging relationship between arbitrary tasks.

3. Viability of the Ptrace Interface

The UNIX† Operating System's *ptrace()* interface was examined to see if it could be extended to fulfill these requirements. One common extension to *ptrace()* is to allow it to attach to a target that is not a direct descendant of the debugger. This requires changes to various system tables so the target will be correctly identified when the debugger performs a *wait()* system call.

Normally, the only information passed to the debugger through *wait()* is the process id of the target and the signal that stopped it. This is sufficient when the only reason to report status to the debugger is the occurrence of a signal. However, additional information needs to be passed when a fork, exec, or other system operation requires the debugger be notified. This information could be passed in the 16 unused bits of the wait structure or obtained by a new *ptrace()* operation.

---

† UNIX is a registered trademark of UNIX System Laboratories, Inc.

The major problem is how to name a particular thread in the target task using the *wait()* and *ptrace()* calls. Using the process id is difficult because the UNIX guest OS associates multiple threads with one process. The naming problem can be avoided by stopping all of the threads when one of them takes a signal. Then any changes to thread state would only modify the thread to which the signal would be delivered. Unfortunately, stopping all of the threads in a task could cause some applications to fail due to timing considerations.

Lastly, using the *ptrace()* interface is only applicable to applications running under the UNIX guest OS. Applications running under other guest OSs cannot be debugged using it as there is no way to name them. These issues prompt consideration of other mechanisms.

4. Debugging the Mach Way

After looking at the *ptrace()* interface, the Mach method of debugging a process was examined. This approach combines Mach primitives with the debug interface provided by the guest OS.

4.1. Using Mach Primitives

The following ports are used by a debugger to debug a Mach program. The port representing the target's task port is used to manipulate the state of the target task. The ports representing the target's thread ports are used to manipulate the state of the target threads. And the ports representing the target's task exception port is used to detect faults that occur while the target is running. To debug a target process the debugger obtains the task port of the target. This is currently done by calling *task_by_unix_pid()*. The task port is then used to obtain the thread ports of the threads executing in the target task. Once this is done, the debugger creates a port and calls *task_set_exception_port()* to register send rights to the port as the target's task exception port.

The exception port is used to send messages to the debugger concerning any machine or software exceptions that occur in the target task. These exceptions include but are not limited to: breakpoints, bad memory accesses, invalid instructions, and other synchronous faults. An exception message has a fixed shape containing the the type of exception as well as a code and a subcode that further define the exception. The thread that generated the exception will remain blocked until a reply message is received. Once a reply is received, the thread will continue execution.

To manipulate the state of the target, a debugger may use the following Mach primitives: the *vm_read()* call is used to read pages in the target task; the *vm_write()* call is used to write pages in the target task; the *vm_protect()* call is used to allow breakpoints to be written to read only pages; the *thread_get_state()* call is used to read the registers and other state information concerning a thread; the *thread_set_state()* call is used to write the above information. The thread state will also include elements that control single step execution and other hardware/OS assistance useful for debugging programs. In addition *task_suspend()*, *task_resume()*, *thread_suspend()*, and *thread_resume()* may be used by the debugger to control the target. [4].

4.2. Combining Mach Primitives with the Guest OS's Primitives

To debug a program, the Mach primitives are used in conjunction with the guest OS's debugging mechanism. When using the guest OS's debugging mechanism, the debugger will receive debug information from two sources: from the source supported by the guest OS, and from the target's exception port.

Each guest OS's interface and semantics will produce its own set of problems. In the UNIX guest OS for example, the *ptrace()* mechanism can be used to manipulate only the immediate child of the debugger. [5].

5. Filling in the Holes

The Mach primitives provide a good mechanism for debugging a program. However, they have no concept of what the guest OS may do that will affect a debug session. For example, the first thing that a guest OS will generally do in a debugging session is load the program into the target's memory. A new mechanism is needed that will provide a general way for a guest OS to notify the debugger that a guest OS specific event has occurred. The approach used is to have the guest OS generate a Mach remote procedure (RPC) call on behalf of the target thread to the debugger when ever one of these events occurs. This is similar to the existing exception mechanism except that the guest OS generates the messages instead of the µkernel.

5.1. The Trace Event

The basic component of this new mechanism is the trace event. A trace event is a guest OS defined action that the target performs or is subjected to, which may be of interest to a debugger. When a trace event occurs, a trace event message containing the details of the trace event is sent to the debugger via an RPC. The shape and contents of a trace event message and the reply to each message are defined by the guest OS generating the message. A guest OS may use a different shape message for each event it generates. A trace event will include at least a reply port as well as the thread and task ports of the thread that generated the event. In addition, a trace event may carry other information relevant to the event. This might include the thread port of a newly created thread, the name of the file containing newly loaded code, or any other piece of relevant information. The data in the reply may be used by the guest OS to modify the behavior of the target process. These messages are sent from the target to the debugger on the trace event port. Replies to these messages must be sent on the port contained in the trace event message.

5.2. The Trace Event Port

Each guest OS will maintain a registration of send rights to a port for each task being debugged. This port is called the trace event port and is used to send trace event messages from the target to the debugger. When a target creates a new task, the guest OS will propagate the registration of the trace event port to the newly created task. This task will then be a target of the same debugger as the task that created it.

5.3. The Trace Event Reply Port

Each trace event message will include a reference to a port that is to be used as a reply port for the message. A target will have one reply port for each thread that is executing within it. It may have additional reply ports if system services are being provided by multiple servers. These reply ports may exist for the life of the debug session or they may only be valid long enough to receive the reply message.

5.4. The Trace Event List

For each target thread a guest OS may maintain a trace event list, a list of trace events that will be sent to a debugger. A debugger will be able to enable or disable events in this list. When a debugger attaches to a target, the event lists for the threads in the target will be empty. The debugger then enables the events it wishes to receive. A trace event message will be sent to the debugger only if the event is enabled in the trace event list and the the target has a valid port

registered as its trace event port. A trace event list is needed by an OS that generates a large variety of events that will only be of interest to a debugger under certain conditions. An event list can be used to facilitate backward compatibility. To do so, a debugger should enable only those events it is capable handling. If in the future more trace events are defined by the guest OS, an existing debugger will not be affected because the new events will not be sent to it. If a guest OS does not maintain a trace event list, all events will be sent to the trace event port if it is set. When a target creates a thread the trace event list of the new thread will be empty. Requests to get and set the elements of the trace event list are made by sending a message to the trace control port.

5.5. The Trace Control Port

Each guest OS will maintain a single trace control port. This port is used by debuggers to send messages to the guest OS. These messages include requests to attach to a task and requests to change the event list of a target thread. Send rights to the trace control port for a guest OS are registered with the net message server.

6. Establishing the Target/Debugger Relationship

Figure 1 describes the messages sent to establish the relationship among the various tasks in a debug session. The following messages will establish a debug relationship.

[1] In order for a guest OS to support trace events, it must register send rights to its trace control port with the naming service of the net message server. If only a single instance of the guest OS is running on the system then a name in the form of TRACE_*guestOS* is sufficient. If more then one instance is running then the name should be in the form TRACE_*guestOS.number*. The name and related port are registered with the *netname_check_in()* call.

[2] To start a debugging session the debugger must determine the name of the guest OS that is running the target. This information may be provided by the user, through the environment, or in a configuration file. Once the name of the guest OS is determined it is used to construct the name of the guest OS's trace control port. The debugger the calls *netname_look_up()* to get the trace control port.

[3] Upon receipt of rights to the trace control port, the debugger calls *trace_request()* to send a message to the guest OS requesting send rights to the task port of the target process. This message will contain the guest OS's identification for the target process, and a reference to the task port of the debugger. The guest OS will either return an error or the rights to the task port of the target. The reference to the debugger's task port is included so that the guest OS can make a decision on whether or not to allow the debug operation to occur. In the case where both the target and debugger are operating under the same guest OS, the guest OS can use the reference to the debugger's task port to determine the debugger's identity. Once the guest OS knows the identity of both the target and debugger, it can determine if the debug relationship is allowable.

[4] Once the rights to the task port are obtained, the debugger will call *trace_set_port()* to send a message to register the trace event port.

[5] The debugger may then call *trace_set_events()* to set the event list for each of the target threads.

[6] The debugger then calls *task_set_exception_port()* to register the task exception port.

Figure 2 shows the tasks and their relationship to each other through the various ports used by the debugger.

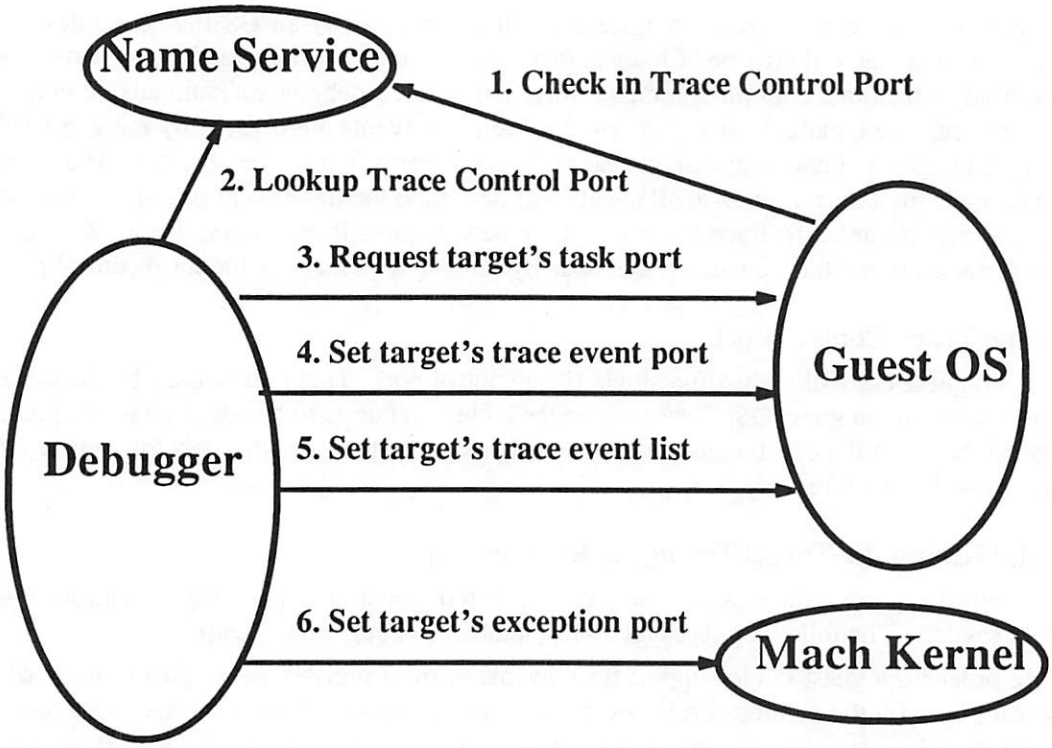


Figure 1.

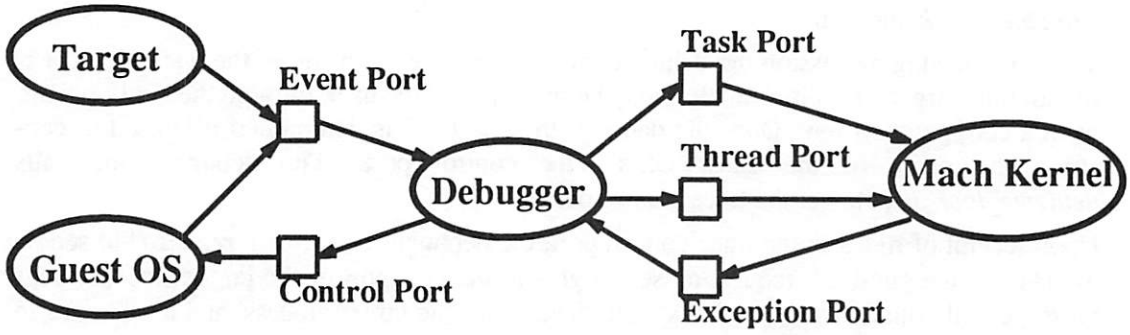


Figure 2.

7. Other Clients of a Debugger

A guest OS is not the only entity that may need to provide information to a debugger. Many languages have special operations that may be of interest. The trace event interface can be used by such a language to communicate with a debugger. For a language to take advantage of this, cooperation between the language's run time module (RTM) and the guest OS will be required. Whenever the the task becomes the target of a debugger the guest OS must pass a reference to the trace event port to the RTM. It must also notify the RTM when it is no longer the target of a debugger. Both of these notification can be delayed until such time as the target generates a language specific trace event. Exactly how the guest OS and RTM communicate will depend on the implementation of the RTM and guest OS.

8. Interface Details

This section describes the ports used by a debugger and the messages that can be sent to them.

8.1. Details on the Trace Control Port

The debugger has send rights to the port registered as the trace control port of the guest OS of the target. Messages sent on the port will be received by the guest OS. These procedures are defined in terms of the Mach Interface Generator (MIG). [6].

8.1.1. Type Definitions

The *trace_target_t* type is the place holder type for guest OS's identifier of a target under its control. Many guest OSs identify a process with a short integer. Some have more complicated schemes. Others only have one program running at a time. By using an array type definition we can accommodate all of these identifiers.

```
type trace_target_t      = array [*:64] of int;
```

The *trace_event_list_t* type is the place holder type for each guest OS's event list. Each guest OS may define a structure that allows the enabling and disabling of each event that the OS can generate. That structure will be passed in place of the *trace_event_list_t* array.

```
type trace_event_list_t = array [*:256] of int;
```

8.1.2. The Trace Request Message

The *trace_request()* RPC is used by the debugger to obtain the task port of the target program. The *trace_control_port* is the port obtained from the naming service of the net message service. The debugger's task port is provided so that the guest OS may make a decision about whether or not the debugger has permission to debug the target. The target identifier is the guest OS's representation of the identity of a process it controls. The *target_task* is returned if the call is successful.

```
routine trace_request(
            trace_control_port     : port_t;
            debugger_task          : task_t;
            target_identifier      : trace_target_t, IsLong;
        out target_task            : task_t);
```

8.1.3. The Trace Set Port and Trace Get Port Messages

The *trace_set_port()* and *trace_get_port()* RPCs are used to set and inquire about the trace event port of a particular task. The *trace_set_port()* RPC establishes a port created by the debugger as the trace event port in the target_task. If another debugger is currently managing the target the old relationship is broken and this one created. This allows one debugger to pass off a target to another debugger. If the value of trace_event_port is the NULL_PORT then the target/debugger relationship is broken and all events are disabled. The *trace_get_port()* returns the trace event port for the target task.

```
routine trace_set_port(
            trace_control_port      : port_t;
            target_task             : task_t;
            trace_event_port        : port_t);

routine trace_get_port(
            trace_control_port      : port_t;
            target_task             : task_t;
        out trace_event_port        : port_t);
```

8.1.4. The Trace Set Events and Trace Get Events Message

The *trace_set_events()* indicates the events that a guest OS will trace. The *trace_get_events()* call returns the events that the guest OS is currently tracing. The *trace_event_list_t* is a structure of a form defined by the guest OS.

```
routine trace_set_events(
            trace_control_port      : port_t;
            target_thread           : thread_t;
            event_list              : trace_event_list_t, IsLong);

routine trace_get_events(
            trace_control_port      : port_t;
            target_thread           : thread_t;
        out event_list              : trace_event_list_t, IsLong);
```

8.2. Details on the Trace Event Port

The guest OS is responsible for propagating send rights of the trace event port to all of the tasks that will need it. In general this will include the target task and any system servers that will generate events on behalf of the target task. The target or server will keep track of the value of the trace event port. If the value is PORT_NULL, no trace event messages will be sent. If the value is not PORT_NULL, trace event messages will be sent if the event is enabled. At some point the debugger may deallocate the trace event port without notifying the guest OS. In this case the RPC will return an error status of SEND_INVALID_PORT. If this occurs the target or server should set the value of the trace of the trace event port to PORT_NULL and disable all events. The format and names of the event messages are determined by the guest OS.

9. Specifics for the UNIX Guest OS

The initial implementation of this mechanism was for the UNIX guest OS. The following are the type definitions and trace event RPCs. Most of the definitions are made in terms of MIG. Routine names will be proceeded with *report_* when called from the guest OS (the client), and with *catch_unix_* when called from the debugger (the server).

9.1. UNIX Trace Type Definitions

The following type definitions are used by the UNIX guest OS interface.

The *trace_target_t* type is replaced by the *pid_t* type as defined by the UNIX guest OS. The *target_identifier* argument of the *trace_request* call will have an argument of a pointer to a

pid_t passed instead of an argument of *trace_target_t* type. The UNIX_TARGET_COUNT is passed as the length of the target_identifier argument.

```
#define UNIX_TARGET_COUNT (sizeof(pid_t)/sizeof(int))
```

The *unix_event_list* structure represents the event list used by the UNIX guest OS. An argument of *unix_event_list_t* is passed as the *event_list* argument to the *trace_set_events()* and *trace_get_events()* call. This type is used instead of the *trace_event_list_t* place holder. The length of the event_list to be passed is UNIX_EVENT_LIST_COUNT. If any boolean member of the *unix_event_list* is set to TRUE the event messages will be sent. In the case of signals, if the signal is a member of the *signal_events* set a trace event will be sent.

```
typedef struct unix_event_list {
        sigset_t        signal_events;
        boolean_t       fork_event;
        boolean_t       exit_event;
        boolean_t       exec_event;
        boolean_t       load_event;
        boolean_t       unload_event;
        boolean_t       system_call_event;
        boolean_t       pthread_create_event;
        boolean_t       pthread_exit_event;
        boolean_t       pthread_context_assignment;
};
#define UNIX_EVENT_LIST_COUNT \
    (sizeof(struct unix_event_list)/sizeof(int))
typedef struct unix_event_list *unix_event_list_t;
typedef struct unix_event_list unix_event_list_data_t;
```

The following MIG structures are used to report the arguments to a UNIX system call and modify its return values. Upon surveying all UNIX system calls it was determined that no system call has more than six arguments. Some UNIX system calls (such as *fork()* and *getpid()*) have two return values. Unfortunately the only documentation on such calls is the UNIX kernel source itself.

```
type syscall_arguments_t        = array [6] of int;
type syscall_return_values_t    = array [2] of int;
```

9.2. The Signal Event

The delivery of a signal implies the termination of a task or the asynchronous calling of a signal handler in a thread. By reporting the delivery of a signal, the debugger gets the opportunity to correct the cause of the signal and allow the target to continue. Or it may allow the signal to be delivered and debugging of the signal handler to commence. Another possibility would be to ignore the signal or deliver it later with the *kill()* call.

This message is sent any time a signal is delivered that is a member of the *signal_events* field of the *unix_event_list*. *Signalin* is the signal that was generated. *Signalout* is the signal that will be delivered. If *signalout* has a value of 0, no signal will be delivered. The thread is blocked until the reply is received. If the reply message has an error status or *signalout* is invalid then

signalin will be delivered.

```
routine signal(
        requestport   trace_event_port   : port_t;
        replyport     clear_port         : port_t;
                      thread             : thread_t;
                      task               : task_t;
                      signalin           : int;
                  out signalout          : int);
```

9.3. The Fork Event

When a fork occurs, the debugger inherits another target program that it may debug. Reporting a fork event allows the debugger to clean up any breakpoints in the parent and child tasks. It also allows the debugger to gain control of the child before it executes any user space code.

Any time a fork occurs and the *fork_event* field of the *unix_event_list* in non-zero, this message is sent. *Child_thread* is the thread port of the newly created thread. *Child_task* is the task port of the newly created task. Both the parent thread and child thread are blocked until the reply is received.

```
routine fork(
        requestport   trace_event_port   : port_t;
        replyport     clear_port         : port_t;
                      thread             : thread_t;
                      task               : task_t;
                      child_thread       : thread_t;
                      child_task         : task_t);
```

9.4. The Exit Event

The exit event allows the debugger to examine the target's state before it is destroyed. This is useful for obtaining a final back trace.

Any time a task exits and the *exit_event* field of the *unix_event_list* is non-zero, this message is sent. The *exit_value* field contains the exit value that will be reported by *wait()* to the parent of the task. The thread and task still retain their states as they were just before the exit occurred. The task will exit when the reply message is received.

```
routine exit(
        requestport   trace_event_port   : port_t;
        replyport     clear_port         : port_t;
                      thread             : thread_t;
                      task               : task_t;
                      exit_value         : int);
```

9.5. The Exec Event

When a target execs a new executable image it, discards the old executable image. This includes any breakpoints that were set in the old image. The exec event allows the debugger to set new breakpoints in the target and complete preparation for debugging a new target.

Any time a task execs and the *exec_event* field of the *unix_event_list* is non-zero, the exec event message is sent. The state of the task and thread will be set to their initial state at the beginning of the new program. If the program was loaded by the OSF user space loader, [7]. the thread's state will be that of the first instruction of the loader. If the program was loaded entirely by the UNIX guest OS, the thread's state will be that of the first instruction of the program. *Argp* is the address in the target task that contains the characters of the arguments array. *Arg_size* is the total size of the array. *Envp* and *env_size* point to and give the length of the environment array in the target's address space. The format of these arrays is a set of characters terminated by a 0 byte followed immediately by the next set of characters, etc. The new image will execute when the reply is received.

```
routine exec(
        requestport  trace_event_port  : port_t;
        replyport    clear_port        : port_t;
                     thread            : thread_t;
                     task              : task_t;
                     argp              : vm_offset_t;
                     arg_size          : vm_size_t;
                     envp              : vm_offset_t;
                     env_size          : vm_size_t);
```

9.6. The Load Event

When the guest OS reports a load event, the debugger will have a chance to load the symbol table of the freshly loaded code. The debugger will also be able to set breakpoints before the new code is executed.

This event is generated by the OSF user space loader after it has loaded one or more related executable images and before the initialization routines of the images are called. The load event message will be sent if the *load_event* field of the *unix_event_list* is non-zero. Execution will continue when the reply is received.

```
routine load(
        requestport  trace_event_port  : port_t;
        replyport    clear_port        : port_t;
                     thread            : thread_t;
                     task              : task_t);
```

9.7. The Unload Event

The unload event provides symmetry with the load event. When the guest OS reports an unload event, a debugger can purge the symbols of the unloaded image from its symbol table. It will also be able to determine which breakpoints will be lost when the code is removed from memory.

This event is generated by the OSF user space loader after it has removed a module from the loaded package table list and before it has removed it from memory. The unload event message will be sent if the *unload_event* field of the *unix_event_list* is non-zero. Execution will continue when the reply is received.

```
routine unload(
        requestport   trace_event_port   : port_t;
        replyport     clear_port         : port_t;
                      thread             : thread_t;
                      task               : task_t;
                      module_id          : int);
```

9.8. The System Call Event

Intercepting system calls allows the debugger to emulate the behavior of the OS. This allows a debugger to run a program that should be run as a setuid program. The debugger intercepts any calls that are affected by the identity of the target and adjusts their results so that the target believes that it is actually a setuid program. In addition, by intercepting system calls, the debugger could create an activity log of an executing program and then allow the user to rerun the program having the system calls return the same results as in a previous run.

This event message is sent before a system call is executed whenever the *system_call_event* field of the *unix_event_list* is non-zero. The *call_number* is defined in <sys/syscall.h>. The *arguments* parameter is an array containing the arguments to the system call. If *fake_call* is returned with a value of false, the system call will execute normally. If *fake_call* is returned with a value of true, the system call will not be performed. In this case *error_value* and *return_values* will be returned to the target program. An *error_value* of zero will cause the *return_values* to be returned to the caller. Any other *error_value* will cause that error to be returned.

```
routine system_call(
        requestport   trace_event_port   : port_t;
        replyport     clear_port         : port_t;
                      thread             : thread_t;
                      task               : task_t;
                      call_number        : int;
                      arguments          : syscall_arguments_t;
                out   return_values      : syscall_return_values_t;
                out   error_value        : int;
                out   fake_call          : boolean_t);
```

9.9. The Pthread Create Event

This event allows the debugger to obtain control of a newly created thread before it executes any code, and gives the debugger the opportunity to set breakpoints or change the initial state of the new thread.

This event will be sent whenever a new pthread is created and the *pthread_create_event* field in the *unix_event_list* is non-zero. *New_thread* is the thread port of the newly create thread. *Thread_name* is the pthread identifier for the thread. There is no guarantee that a particular pthread will always use the same kernel thread to execute its instructions. *Thread_argument* is

the argument passed to the first procedure of the new thread. Both the current thread and the new thread will be blocked until a reply is sent.

```
routine pthread_create(
        requestport   trace_event_port   : port_t;
        replyport     clear_port         : port_t;
                      thread             : thread_t;
                      task               : task_t;
                      new_thread         : thread_t;
                      thread_name        : pthread_t;
                      thread_argument    : vm_offset_t);
```

9.10. The Pthread Exit Event

This event will give the debugger access to the state of a thread just before it exits. This is useful for obtaining a final back trace.

This event message will be sent whenever a pthread exits and the *pthread_exit_event* field in the *unix_event_list* is non-zero. The *thread_status* is the status passed to *pthread_exit()*. The thread will exit when the reply is received.

```
routine pthread_exit(
        requestport   trace_event_port   : port_t;
        replyport     clear_port         : port_t;
                      thread             : thread_t;
                      task               : task_t;
                      thread_name        : pthread_t;
                      thread_status      : vm_offset_t);
```

9.11. The Pthread Context Assignment Event

As mentioned while discussing the pthread create event, there is no guarantee of a one to one correspondence between a pthread and a kernel thread. In a single thread implementation of pthreads this message will be sent for every context switch. In an implementation that does guarantee a one to one correspondence between pthreads and kernel threads, this message will never be sent. Other implementations will send this message whenever the pthread that is executing in a kernel thread changes. This message will only be sent if the *pthread_context_assignment* field in the *unix_event_list* is non-zero. *Old_pthread* is the pthread that is losing the use of the *thread*. *New_pthread* is the pthread that is gaining the use of *thread*. If either of these pthreads is the value NO_PTHREAD then the parameter does not refer to any pthread.

```
routine pthread_context_assignment(
        requestport   trace_event_port   : port_t;
        replyport     clear_port         : port_t;
                      thread             : thread_t;
                      task               : task_t;
                      old_pthread        : pthread_t;
                      new_pthread        : pthread_t);
```

10. Example of a Unix Debugger

The following routine, *unix_target()*, will establish a debug relationship with a target application running under the UNIX guest OS. The *unix_target()* routine will then dispatch events to the appropriate MIG generated routines, *exc_server()* and *unix_trace_server()*. These routines will then make call backs to the *catch_exception_raise()*, *unix_catch_signal()*, *unix_catch_fork()*, and *unix_catch_exec()* routines.

```
unix_target(
    pid_t                   pid
) {
    port_t                  control_port;
    port_t                  event_port;
    port_t                  exception_port;
    port_t                  client_ports;
    port_t                  target_task;
    kern_return_t           ret;
    thread_array_t          thread_list;
    unsigned int            count;
    unsigned int            ii;
    unix_event_list_data_t  event_list;
    struct dummy_message    request;
    struct dummy_message    reply;

    /* allocate ports needed to receive events and exceptions */
    ret = port_allocate(task_self(), &event_port);
    ret = port_allocate(task_self(), &exception_port);
    ret = port_set_allocate(task_self(), &client_ports);
    ret = port_set_add(task_self(), client_ports, event_port);
    ret = port_set_add(task_self(), client_ports,
        exception_port);

    /* initialize the event list structure */
    bzero(&event_list, sizeof(event_list));
    sigfillset(&event_list.signal_events);
    event_list.fork_event = TRUE;
    event_list.exec_event = TRUE;

    /* get the task port of the target task */
    ret = netname_look_up(name_server_port, "",
            "TRACE_UNIX", control_port);
    ret = trace_request(control_port, task_self(),
            &pid, UNIX_TARGET_COUNT, target_task);

    /* setup the debugger/target relationship */
    ret = task_suspend(target_task);
    ret = trace_set_port(control_port, target_task, event_port);
    ret = task_threads(target_task, &thread_list, &count);
    for (ii = 0; ii < count; ii++)
        ret = trace_set_events(control_port, thread_list[ii],
```

```
                &event_list, UNIX_EVENT_LIST_COUNT);
    ret = task_set_exception_port(target_task, exception_port);
    ret = task_resume(target_task);

    /* dispatch events from the target to the MIG servers */
    for(;;) {

        /* wait for and event message or exception message */
        request.Head.msg_local_port = client_ports;
        request.Head.msg_size = sizeof(request);
        ret = msg_receive(&request.Head, MSG_OPTION_NONE, 0);

        /* pass the message to the correct MIG server routine */
        if (request.Head.msg_local_port == exception_port)
                exc_server(&request, &reply);
        else if (request.Head.msg_local_port == event_port)
                unix_trace_server(&request, &reply);

        /* send the reply back to the target */
        reply.Head.msg_local_port = request.Head.msg_local_port;
        reply.Head.msg_remote_port =
                request.Head.msg_remote_port;
        ret = msg_send(&reply.Head, MSG_OPTION_NONE, 0);
    }
}

/* exc_server's call back for exceptions */
catch_exception_raise(
    port_t          exception_port,
    port_t          clear_port,
    thread_t        thread,
    task_t          task,
    int             exception,
    int             code,
    int             subcode
) {
    /* code to handle exceptions */
}

/* unix_trace_server's call back for signal events */
unix_catch_signal(
    port_t          event_port,
    port_t          clear_port,
    thread_t        thread,
    task_t          task,
    int             signalin,
    int             *signalout
) {
    /* code to handle signals */
```

```
}

/* unix_trace_server's call back for fork events */
unix_catch_fork(
    port_t          event_port,
    port_t          clear_port,
    thread_t        thread,
    task_t          task,
    thread_t        child_thread,
    task_t          child_task
) {
    /* code to handle forks */
}

/* unix_trace_server's call back for exec events */
unix_catch_exec(
    port_t          event_port,
    port_t          clear_port,
    thread_t        thread,
    task_t          task,
    vm_offset_t     argp,
    vm_size_t       arg_size,
    vm_offset_t     envp,
    vm_size_t       env_size
) {
    /* code to handle execs */
}
```

11. Limitations

When using this interface there are several issues to keep in mind. These include process identity and UNIX signal delivery.

11.1. Identity

One such issue is that of process identity. The µkernel has no concept of identity. Only the guest OS controlling a process knows the identity of the process. However, the guest OS does not control all access to a task. If another process has send rights to the task kernel port of a process, the guest OS cannot change the identity of the process. Doing so would grant access to the new identity to the other process holding rights to the target's task kernel port. Furthermore, the guest OS cannot determine which task has send rights to the port and what its identity is. For the UNIX guest OS, if a debugger is attached to a process, then the process cannot be allowed to change its identity by executing a setuid program. This is true even if the other task holding send rights to the target's task kernel port has appropriate privileges.

11.2. UNIX Signal Delivery

In the past, UNIX debuggers received notification of a signal near the time it was raised. The debugger was notified of the signal regardless of whether it was ignored, held, handled or left at the default action. This allowed the debugger to share the same controlling terminal as the

target. When the user hit the interrupt key, the debugger noticed because the target stopped. This stop occurred regardless of what the target wanted to do with the signal.

The trace mechanism only notifies the debugger when a signal is delivered. This allows the target to run more closely to the way it would when running without a debugger attached. However, if the target is ignoring the interrupt signal, the debugger will not notice when the user hits the interrupt key. To solve this problem, the debugger will have to arrange another avenue of input from the user for itself.

12. Conclusions

This interface provides a mechanism for debugging programs that can be easily extended in the future. Currently, only the UNIX OS interface has been designed. It was recently extended to deal with the OSF user space loader and pthreads. Other new functionality in the guest OS may be added with equal ease. This can be done in a manner compatible with existing debuggers through careful use of the event lists. By having the debugger clear the data of an event list and then turning on the events it desires, it will never receive events it does not understand.

The behavior of target programs can be kept very close to their behavior when running without a debugger. Events are reported at as late a moment as possible. Timing problems are reduced by enabling or disabling individual events. The costs will only be paid for events that the debugger has actually enabled. A program that has a debugger attached to it but with no events enabled should run with no changes in behavior.

By involving the net message server in the arrangement of debugger and target relationships it becomes possible for a program running on one machine to debug a program on another machine. In the future, if the problem of verifying identity between different guest OSs is solved, it will be possible for a debugger running under one guest OS to debug a target running under a different guest OS.

Acknowledgements This work benefited from the conversations I have had with Nawaf Bitar, Tracy Hoover, Dave Leblang and Dan McCue.

References

1. Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, "Mach: A New Kernel Foundation for UNIX Development," *USENIX Conference Proceedings*, July 1986.
2. David Golub, Randall Dean, Alessandro Forin, and Richard Rashid, "Unix as an Application Program," *USENIX Conference Proceedings*, Anaheim, CA, June 1990.
3. Hector Garcia-Molina, Frank Germano, Jr., and Walter H. Kohler, "Debugging a Distrbuted Computing System," *IEEE Transactions on Software Engineering*, vol. SE-10, No. 2, March, 1984.
4. David L. Black, David B. Golub, Karl Hauth, Avadis Tevanian, and Richard Sanzi, "The Mach Exception Handling Facility," *CMU Technical Papers*, April 1988.
5. Deborah Caswell and David Black, "Implementing a Mach Debugger For Multithreaded Applications," *USENIX Conference Proceedings*, Washington, D. C., January 1990.
6. Richard R. Draves, Michael B. Jones, and Mary R. Thompson, "MIG - The MACH Interface Generator," *CMU Technical Papers*, August 1987.
7. Larry W. Allen, Harminder G. Singh, Kevin G. Wallace, and Melanie B. Weaver, "Program Loading in OSF/1," *USENIX Conference Proceedings*, Dallas, TX, January 1991.

Kernel Support For Network Protocol Servers

Franklin Reynolds
fdr@osf.org

Jeffrey Heller
jeffreyh@osf.org

Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142

Abstract

The current Mach 3.0 network device interface provides support for user space protocol servers. This support includes mechanisms for user space device drivers and a message based device interface including a programmable packet filter for demultiplexing network packets. The flexibility and other benefits of these features are not without cost. The goal of our work is to improve network performance, cpu utilization, OSF/1 network driver compatibility, and support for protocol server compression while avoiding any significant perturbation of the Mach 3.0 interfaces. The design and implementation of a shared memory communication channel between the kernel interrupt handler and the user space network protocol server is described. A variety of performance measurements are reported and compared to measurements made on similar systems. Some of the implications of protocol servers on hardware and software architectures are discussed.

Introduction

Traditional operating systems kernels such as BSD 4.3, Mach 2.5 and OSF/1 provide all OS functionality, including networking, as part of a single integrated kernel. When network packets arrive the networking protocol code is able to process this data directly. In some cases no data copies are necessary. For instance, for packets with network specific information such as ARP and routing information, and only one copy is needed from the kernel's address space to the users buffer for many other types of incoming Packets.

With the advent of Mach 3.0 and the BSD single server [1] the interactions between the network device driver and the network protocols changed. Most BSD functionality, including network protocols, was evicted from the kernel and into a user space server. With the separation of BSD from Mach, device drivers could no longer assume the presence of BSD services. The ability to share information and state between the network device driver and the network protocols was eliminated by the separation by of the address spaces. A new device interface was created to allowing the kernel to export device services to Mach tasks. Most if not all device drivers and the clients of the device driver services written for these traditional operating systems must be modified to work in the new Mach 3.0 environment.

The network device interface for Mach 3.0 is message based. Network packets arrive, are copied into a message and sent to servers listening for packets via IPC. There is a thread that runs as part of the BSD single server which receives these messages. An mbuf is pointed at or wrapped around each message then fed to the network protocol code and processed in the same manner it would have been in a traditional operating system kernel. The kernel exports a programmable packet filter as part of the device interface [2]. The packet filter provides packet shedding, demultiplexing and duplication. This filter interprets a simple, stack oriented language. Filter scripts can be created and installed by user tasks.

To use this new interface and kernel, a network device driver of a BSD or OSF/1 origin requires many changes. For instance, the original network drivers assumed the existence of mbufs and had explicit knowledge of the supported protocol families. The Mach 3.0 drivers do not have BSD or protocol family dependencies. Functionality such as the first level of network packet demultiplexing and filtering is implemented as a packet filter script. Instead of putting the data into some space in the kernel that the driver and the protocol code know about, a message must be put together, data copied into it and sent off to the network

code. This method adds buffer copies and context switches to the cost of receiving a network packet. In the case of small packets, a message based interface adds significant overhead relative to monolithic systems. The 3.0 network device architecture is powerful and flexible and complex. This power and flexibility is not without cost.

Mach 3.0 has experimental support for user space device drivers. There is a reference implementation of an ethernet driver for the PMAX in the 3.0 distribution. The user space ethernet device driver work was motivated by, among other things, the need for better performance. One limitation of this approach is that a single network server is favored over all others. Only one task has direct access to the network driver. Only that task gets the performance advantage. Other tasks wanting information off the wire must arrange with the favored task for it to pass packets they are interested in on to them. Perhaps more important is that some hardware platforms lack the necessary features to make user space device drivers practical or efficient.

The goal of our work is to improve the network performance and efficiency of the OSF/1 single server. We do this by attempting to reduce the number of context switches, data copies and scheduling interactions during message processing. We hope to improve OSF/1 monolithic kernel and single server network driver compatibility by increasing the reuse of OSF/1 driver and low level protocol code. This would reduce the cost to vendors of migrating from monolithic kernel technology to micro-kernel based technology. In addition we attempt to avoid significant perturbation of the Mach 3.0 kernel interfaces.

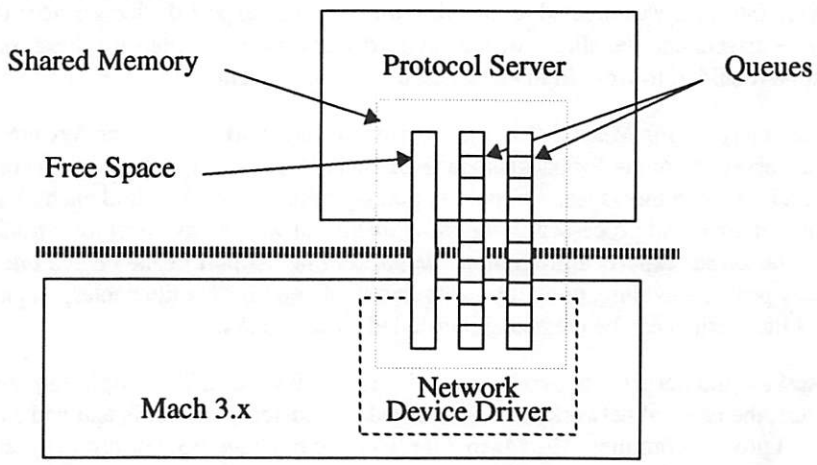
A Shared Memory Device Architecture:

We have developed a experimental network device driver framework that attempts to address our performance, portability and compatibility requirements. Our framework incorporates the use of a shared memory communication channel between the device interrupt handler and the network protocol servers. Our design relies on two kernel extensions.

1) The ability to create shared memory between an in-kernel device driver and a user space task

2) The ability of a kernel interrupt handler to perform a counting unblock of a user space thread.

In this approach the network device driver shares a region of memory with the protocol server. Both the driver and the server have read and write access to the shared memory. Mach 3.0 already defines a dev_map() service that can be trivially extended to provide the desired semantics. The shared memory consists of queues, free space for network buffers and a few shared variables. Queues are associated with devices, protocol families and other services. These queues provide the primary channel for communication between devices and protocols.



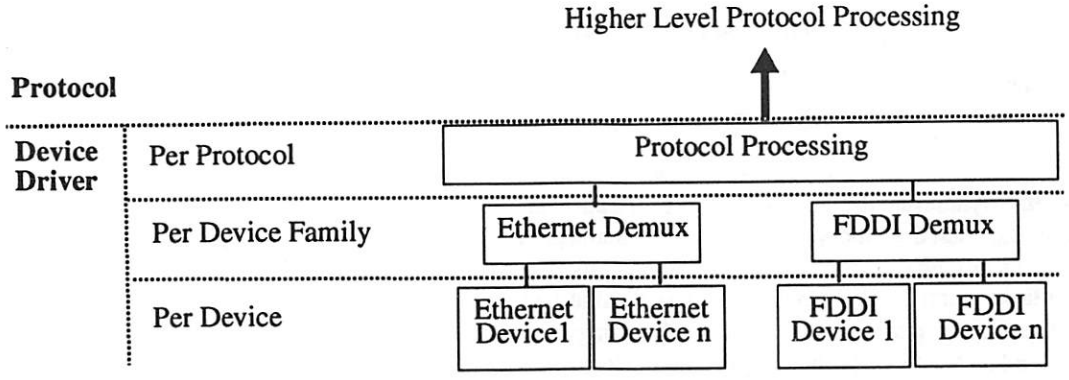
The design is derived from OSF/1 which, though a monolithic kernel, has a similar design. The network interrupt handlers communicate with the protocol threads by placing network buffers on appropriate protocol

queues and then, if necessary, unblocking a network thread. Our approach, of enhancing and transparently extending the functionality of extant abstractions increases OSF/1 device drive compatibility and the reuse of OSF/1 code within the server.

Device Driver and Protocol Server Interdependency:

The OSF/1 device drivers execute in the same context as the protocols. The Drivers exploit apriori knowledge of the framework and the configured protocol families to improve performance. Regardless of the truth and beauty of this issue, the practical consequence in this case is that the network device driver understands the network buffer structures and have some apriori knowledge of the configured protocol families. In the interest of OSF/1 compatibility we BSD 4.3 Reno mbufs, clusters and Stream mblks. BSD 4.3 mbufs are fairly general memory descriptors and have been used extensively. In fact, OSF/1 is able to transparently and inexpensively layer mblks on top of mbufs. Protocol frameworks that use alternative data structures, such as the x-Kernel, which do not wish to adopt mbufs can always make the relatively simple changes to the device drivers necessary to use their own buffer descriptors.

Knowledge of the configured protocol families allows early packet filtering. This is currently done in the OSF/1 device interrupt routine. To take advantage of this will require new protocols to add specific code in the device interrupt handler. In the worst case this will mean a new version of all network device drivers for each new family of protocols. Fortunately, new protocols are relatively uncommon and in most cases the necessary driver modification s are trivial. The first level of packet filtering could be evicted from the kernel but it would, without a doubt, slow some things down. In particular, protocol selection by the driver allows us to support multiple protocol servers without paying the cost of indirection through a demultiplexing server. Another advantage of doing the first level of demux inside the interrupt handler is that corrupt packets can be shed without delay. This improves the robustness of the system by preventing corrupt packets from tying up resources (buffers and CPU cycles) that could be used to handle real data.



The figure above illustrates the hierarchy of packet processing within the device driver. A the lowest level there is per device, usually per network device controller, processing. Device errors would typically be handled at this level. At the next level, the Family level, processing that is specific to ethernet but independent of the individual manufactures versions of ethernet, takes place. For example, the ethernet header is examined for supported protocol types. Packets with bad types, such as test protocols, are dropped. The top layer is entirely hardware independent.

Shared Memory & Synchronization:

Mach 3.0 exports a dev_map() interface. This interface did not change. The original device pager that is established via dev_map() calls dealt with physical pages. A small change to the implementation of the device pager allowed it to handle virtual address. The dev_map() service in conjunction with vm_map() are sufficient to build the shared memory window.

While it is possible to share memory between the kernel and a user task, it is very difficult on some hardware platforms to make the virtual addresses of the shared region align. This alignment, or rather the lack of alignment of the virtual addresses of the shared memory in the kernel and the protocol server pose some problems. Pointers and data references are usually absolute virtual addressees rather than offsets from a base register or segment. As a consequence shared pointers only work without some sort of conversion when the shared memory is mapped to the same location in each virtual address space. Our strategy is to assume that the shared memory structures and pointers should correct for the protocol server and appropriate conversions will be made in the kernel. Translations are done in the driver to localize and minimize the impact on the code base.

Consider the example of a queue of buffers in shared memory that does not align. For the purposes of this example the kernel maps the window at 0xc000000 and the server maps the window at 0x6000. Assume that a shared memory queue begins at the beginning of the window. Both the kernel and the protocol server have non-shared memory pointers that refer to the shared memory queue control block. The value of the server's queue pointer is 0x6000 and the value of the kernel's pointer is 0xc000000. The queue points to a buffer in shared memory. The buffer is offset 0x100 bytes from the beginning of the shared memory window. The value of the queue's pointer to the buffer is 0x6100. Server code can use the queue pointer in a natural fashion. The kernel must convert 0x6100 (which in the kernel's context may refer to almost anything) to 0xc00100 before it can be used as a pointer.

Protocol to interrupt (PTOI) and interrupt to protocol conversion macros have been defined.

```
#define PTOI(addr) (((addr)) ? (unsigned int)(addr) +
                    (unsigned int)plus_shm_offset - (unsigned int)minus_shm_offset : 0 )
#define ITOP(addr) (((addr)) ? (unsigned int)(addr) -
                    (unsigned int)plus_shm_offset - (unsigned int)minus_shm_offset : 0 )
```

Note the test for 0. This is necessary to avoid converting NULL pointers to non-NULL pointers. While the cost of these conversions are not dramatic they could be avoided completely on friendly hardware.

We need to synchronize access to the shared memory between threads on multiple processors and interrupt handlers. Synchronizing access by the user space protocol servers and the kernel space device driver to data in the shared memory present some difficulties. Typically, unprivileged programs use a system call or use either atomic operations such as compare-and-swap instructions to synchronize with the kernel. Privileged tasks, such as the kernel, frequently disable interrupts to synchronize with interrupt handlers. This is because the naive use of locks without disabling interrupts, even spin-locks, to synchronize with interrupt handlers has deadlock problems [3]. The performance characteristics of system calls make them unsuitable for our purposes. In the absence of hardware support across the majority of interesting platforms, a portable software synchronization method must be devised. In the absence of the ability to disable interrupts another method must be employed to prevent the interrupt handler from waiting forever for a lock.

It has been asserted that wait-free synchronization primitives can be engineered with an atomic compare-and-swap instruction [4]. Presumably these primitives would be suitable for synchronizing with interrupt handlers. However, many hardware platforms do not provide hardware support for atomic operations such as compare-and-swap or fetch-and-theta. In fact, there are popular hardware architectures that do not provide even a simple atomic test-and-set instruction. Hardware that allows unprivileged tasks to disable interrupts, a frequently used technique for synchronizing with interrupt routines, are even less common. Because of the general lack of adequate hardware support we chose to provide a software solution. Our hope is that the abstractions will prove amenable to optimization on friendly hardware platforms.

Our queue and the buffer allocation abstractions provide wait-free operations to the device driver's interrupt routines. The queue and buffer management routines use a low level software mutex to implement critical sections. This mutex can be replaced or optimized on friendly hardware platforms. Critical sections are

used to coordinate access to the new data structures necessary to implement the queue the various wait-free operations. There is a per queue conditional variable used to indicate that the agent that removes buffers from the queue has blocked as well as a new mechanism for unblocking threads.

Software Mutex:

We chose to implement a software mutex algorithm to serve as the basis for the higher synchronization protocols. A software approach has the twin advantages of being very portable and amenable to hardware dependent optimizations. We were also interested in assessing the cost of software synchronization algorithms.Software algorithms for mutual exclusion [5] are designed for coordinating multiple processes - not processes and interrupt processing routines. Interrupt handlers have different execution characteristics from processes. Generally speaking, device drivers do not like long busy - wait or spin loops When synchronizing with device drivers, monolithic kernels usually take advantage of privileged instructions to disable and re-enable interrupts. Unprivileged tasks, such as the protocol server must find another way to synchronize with interrupt processing routines.

We have designed an asymmetric variation of Peterson's [6] mutual exclusion algorithm. Descriptions and analysis of the original algorithm can be found in [5],[6].The only significant hardware assumptions we make are integer (32 bit) sized atomic loads and stores. This maximizes the hardware independence of the code. The important characteristic of this variant of the algorithm is that the interrupt side does not spin indefinitely, instead it gives up after of couple of attempts to get the lock. This is required in order to remove the possibility of deadlock. Otherwise deadlock could occur if the device driver interrupted and attempted to acquire the mutex immediately after the protocol server acquired the mutex.

Protocol Side:

```
protocol = TRUE;                    /* announce intent to acquire mutex*/
turn = INTERRUPT;                   /* give the interrupt routine a chance */
while ( interrupt == TRUE && turn == INTERRUPT ) {
        spin() ;                    /* wait if the interrupt routine got there first */
}
< critical section >
protocol = FALSE;                   /* renounce intent to enter and exit*/
```

Interrupt Side:

```
interrupt =TRUE;                    /* announce intent to acquire mutex*/
turn = PROTOCOL;                    /* give the protocol task a chance */
if ( protocol == TRUE && turn == PROTOCOL) {
        interrupt = FALSE;          /* if the protocol routine got there first */
        return(FALSE);              /* renounce intent to enter and exit */
}
< critical section >
interrupt = FALSE;                  /* renounce intent to enter and exit*/
```

Note the asymmetry of the algorithm. The Protocol side of the algorithm is identical to Peterson's original. The protocol or single server is permitted to wait (by spinning) for the interrupt processing to complete. On the other hand, the interrupt handler gives up almost immediately rather than spinning. This removes the risk of deadlocking on the mutex. After the interrupt handler relinquishes its attempt to acquire the first mutex it is free to attempt to acquire another mutex or take other action.

Queues

OSF/1 1.0 already uses queues as the communication channel between the network device driver and the protocol threads. Queue initialization, lock, enqueue, dequeue and flush operations are already defined. We have attempted to preserve these abstractions. While we were largely successful some trivial side effects of our different implementation strategy are visible.

Access to the original OSF/1 queues is synchronized by simple locks. These are spin locks that depend on the ability of the protocol code to disable interrupts in order to synchronize with device drivers. The protocol server, as a unprivileged task, does not have the ability to disable interrupts. Consequently the locks for the shared memory queues are built from the software mutex algorithm described earlier.

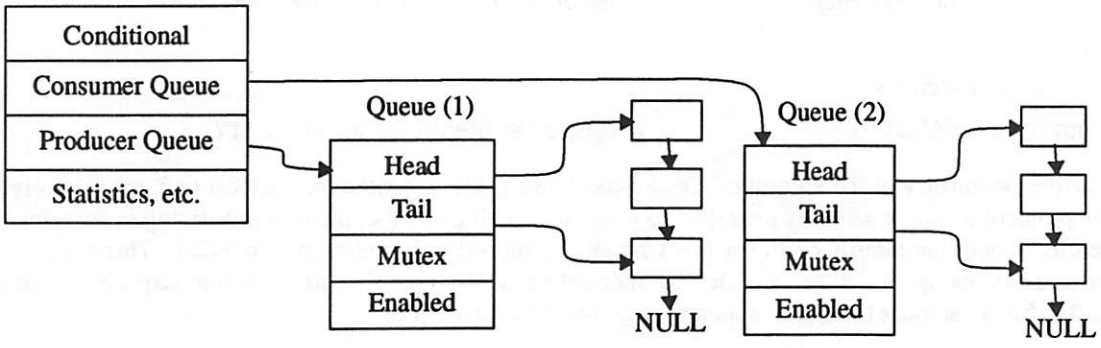
Another consequence of the protocol code executing in a user task is that its execution can be pre-empted for long periods of time. The naive implementation of locks and queues using the software mutex can be problematic if the protocol server blocks or is pre-empted. It is important for good performance that the queues used for sending buffers to protocol stacks be available to the driver. If the queue is unavailable the driver can drop the packet while trusting to the higher level protocols to generate a retry. There are a variety of reasons why this can happen in the OSF/1 monolithic kernel today. However, dropping packets almost always has a bad effect on performance and should not be done lightly.

In some situations the driver **must** be able to successfully enqueue a buffer to avoid leaking resources. It is possible that the device driver is finished with an outbound buffer but cannot successfully return it to free space. This can occur if the free space is locked by the garbage collector or if the buffer is complex such as a chain of buffers or a buffer containing a pointer to a cluster. When the driver decides that a buffer is too complex to free it enqueues the buffer on a deferred-free queue. The protocol pulls buffers off of this queue and frees them at a more leisurely pace. The driver must be able to enqueue the complex buffers onto the deferred-free queue in order to guarantee that the buffer will be eventually freed and not forgotten.

The problem is: design a multi-processor safe queue that is always that is essentially wait-free to the device driver without the hardware support to disable interrupts or perform complex atomic operations.

To accomplish this we define a shared memory queue. There are two types of shared memory queues, IN queues and OUT queues. The Consumer (whatever performs dequeue operations on a particular queue) of an IN queue is the protocol server. The Consumer of an OUT queue is the device driver. The asymmetric nature and needs of the different Consumers dictates an asymmetric implementation of the enqueue and dequeue operations. Each queue is actually composed of two separate queues. There is pointer for the Producer and the Consumer indicating which sub-queue is being used. An enabled flag associated with each sub-queue. is used by the enqueue and dequeue operations to preserve queue order. Each IN or OUT queue has a conditional variable used to indicate that the Consumer for that queue is idle and must be restarted in order to resume dequeue operations.

IN/OUT Shared Memory Queue



Coordinating the use of the sub-queues is a bit complicated. In addition to the software mutex protocol there is a condition variable protocol, a queue ordering protocol as well as the enqueue and dequeue protocols. A conditional variable is associated with each shared memory queue. The Producer reads the condition variable and the Consumer writes it. The Producer tests the condition and if the condition is TRUE then a device event is generated. The Consumer sets the condition to TRUE if there is nothing to do. Then it checks again, in case new work arrived in the interval between the time it last checked for work and when the condition was set. If there is still nothing to do then the Consumer becomes idle. Otherwise the condition is set to FALSE and the Consumer does the appropriate work.

It is possible for the Producer to detect that the condition is TRUE before the Consumer has actually become idle. In order to avoid losing device events the device event service counts events. In our prototype device events are implemented using a version of thread_resume() that counts the resumes as well as the suspends. If the Producer generates a device event before the Consumer attempts to block then when Consumer eventually does attempt to block it will immediately return instead. The condition variable protocol, in conjunction with device events, provides a mechanism for Producers to activate idle Consumers.

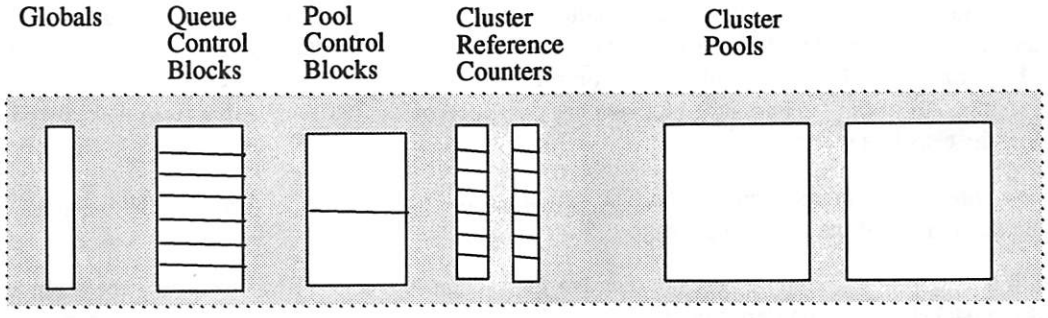
The Enable protocol is used to guarantee that buffers are dequeued from the set of sub-queues in the order that they were enqueued. The Producer only uses the Enabled flag when it discovers that its current sub-queue is locked by the Consumer. In that case it locks the other sub-queue and enables it. The Producer never changes its current sub-queues unless it was blocked. When that occurs it sets the Enabled flag on the new sub-queue to indicate that it successfully changed sub-queue and enqueued a buffer. The Consumer only references the Enabled Flag in the event of a sub-queue becoming empty. When the Consumer's current sub-queue becomes empty the Consumer checks the Enabled Flag of the other sub-queue. If the Producer has successfully changed sub-queues and enqueued at least one buffer the Enabled bit will be TRUE. In that case the Consumer changes its sub-queue, sets the Enabled but to FALSE and dequeues the buffer.

Free Space:

The design of the shared memory queues is complex. Synchronizing access by interrupt handlers and unprivileged tasks to shared memory is difficult. The same difficulties arise in the design of shared buffer space management (allocate, free, garbage collection). An additional source of complexity is both Device drivers and Protocol servers need to allocate as well as free buffers.

A solution to the problem is to divide the buffer free space into multiple pools. All of the control structures are duplicated for each pool. This includes statistics, reference counter arrays, and other control information. Most of the control information is located within a pool control block. Access to each pool is mediated by a Mutex variable. When the device driver needs a buffer it can query each pool. If it is unable to lock the pool or if the pool is empty it can go on to the next pool. If there are no buffers available then the device will discard the packet. This strategy allows us to reuse the bulk of the OSF/1 mbuf and cluster code.

Shared Memory Window



One side effect of multiple free pools is that the buffers need to be returned to the pools from which they were allocated in order to be coalesced into clusters. This is not difficult since each pool is composed of a

contiguous region of memory. The correct pool can be deduced from the address of the buffer. Note the fact that the cluster pools are adjacent in shared memory. This simplifies and improves the performance of the mbuf to pool (mtop) calculations before the original mbuf to cluster number (mtocl) calculation can be performed.

When the server is finished with a buffer or cluster the resource is returned to the appropriate free pool and the reference counters are decremented. When the device driver is finished with a buffer it attempts to return it to the appropriate free pool. If the attempt fails, either because the pool was locked or the buffer was complex (a buffer chain, special free requirements, or a buffer with associated external memory) the interrupt routine places the buffer on the deferred-free queue and the protocol server eventually returns the buffers to the correct free pools.

Sometimes a thread in the OSF/1 monolithic kernel goes to sleep because there are no free buffers available. When a buffer is freed any threads blocked waiting for buffers are receive a wakeup(). OSF/1 uses the global variable "m_want" as a conditional to indicate that there are sleeping threads in need of buffers. Since our design needs to be able to free buffers from protocol or interrupt level processing m_want was moved to shared memory. The server sets m_want to TRUE if any networking threads block in need of buffers. The kernel never changes the state of m_want. This permits m_want to be referenced without first acquiring a lock. After each successful release of a buffer by the kernel m_want is tested. If it is TRUE then the network threads are unblocked. The first of these threads resets m_want to FALSE.

There is an interesting trade-off to be made when choosing the number of pools. Assuming the total amount of free space is constant, a large number of pools decreases the amount of free space controlled by any single mutex. This means that during the time the Protocol holds a particular mutex it is restricting access to a smaller portion of the free space. This gives the Device a better chance of finding a pool with some free space in it. This is a Good Thing. On the other hand, the fragmentation of the free space into many pools will have a tendency to limit garbage collection. It would be problematic to join data into contiguous chunks spanning pool boundaries. Smaller pools also increase the chances that multiple pools will have to be checked before any free space can be found since any individual pool will be more quickly exhausted.

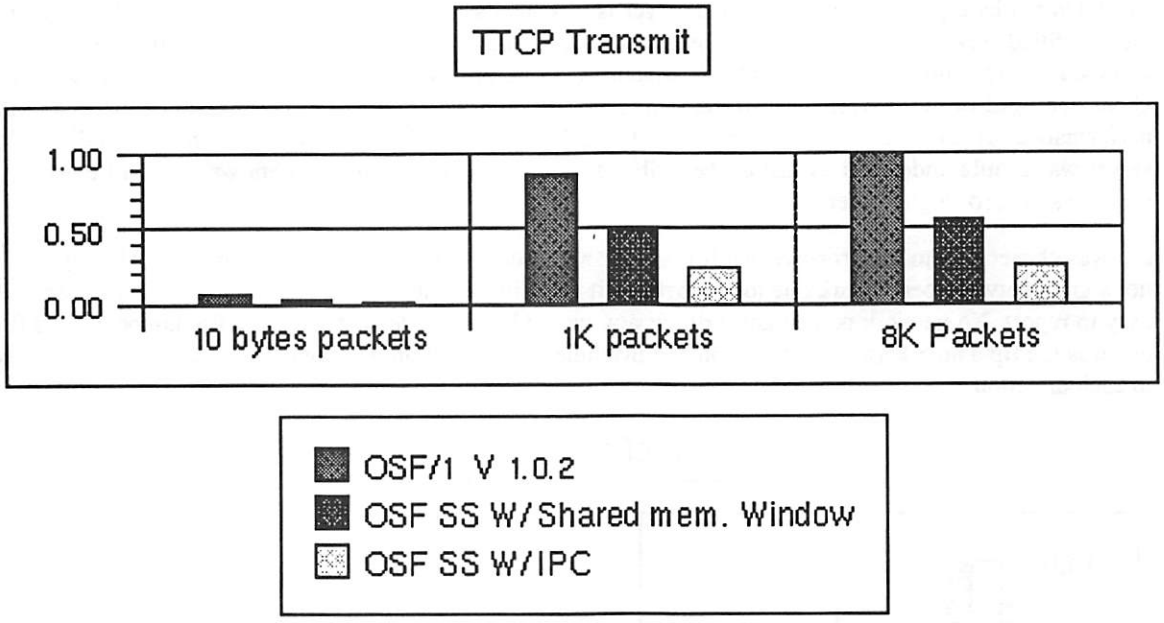
Network Performance:

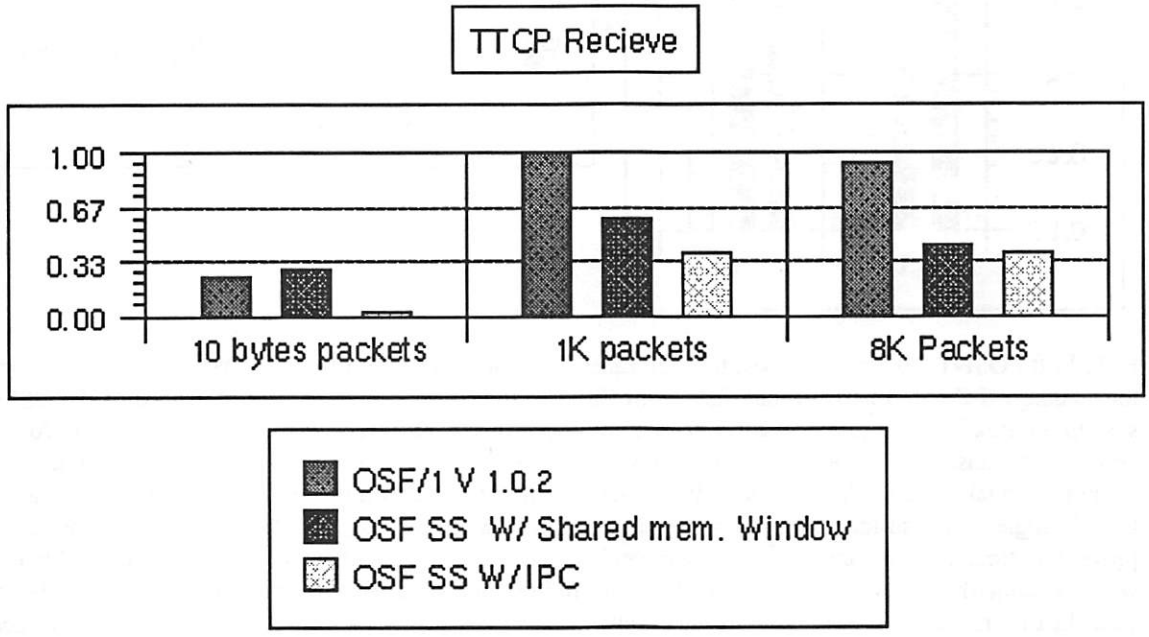
The goal of our performance testing plan is to understand the performance and the cost of this architecture. All of our measurements were done on the same Hp Vectra 386 based computer with 16 megabytes of memory, a western digital wd80013EBT networking card, and an OSF/1 1.0.2 binary set. The only difference between the tests was in the kernel (and single server for the two Mach 3.0 based system) chosen at boot time. The systems that were measured were OSF/1 V1.0.2, the second pre-alpha release of the OSF/1 single server, and that same single server with only the changes described in this paper.

All of the tests were conducted between the 386 machine and a 68040 based NeXT Station. The NeXT has sufficiently greater ethernet performance than the 386 platform that we have confidence that it was never a limiting factor in our tests. As the implementation of all three versions of networking code were at least 90% the same, all measured differences between systems were due to the changes at the device interface layer. The limited number of variables in the performance tests give us give us confidence in our analysis of the results. All of our performance numbers are presented on the bar graphs in a form normalized to the fastest of the tests in a series.

We have chosen to run three benchmarks. The ttcp program - version 1.10 , ftp and the dot benchmark provided in version 1.2 of the X11perf package.

Our first test, ttcp is a streaming network test. No file system interaction happens during the tests, and the size of the packets can be chosen. Six repetitions of transmit and receive tests were done for three different packet sizes. We have chosen a small 10 byte packet size, an average 1K packet size and a larger 8K packet size.

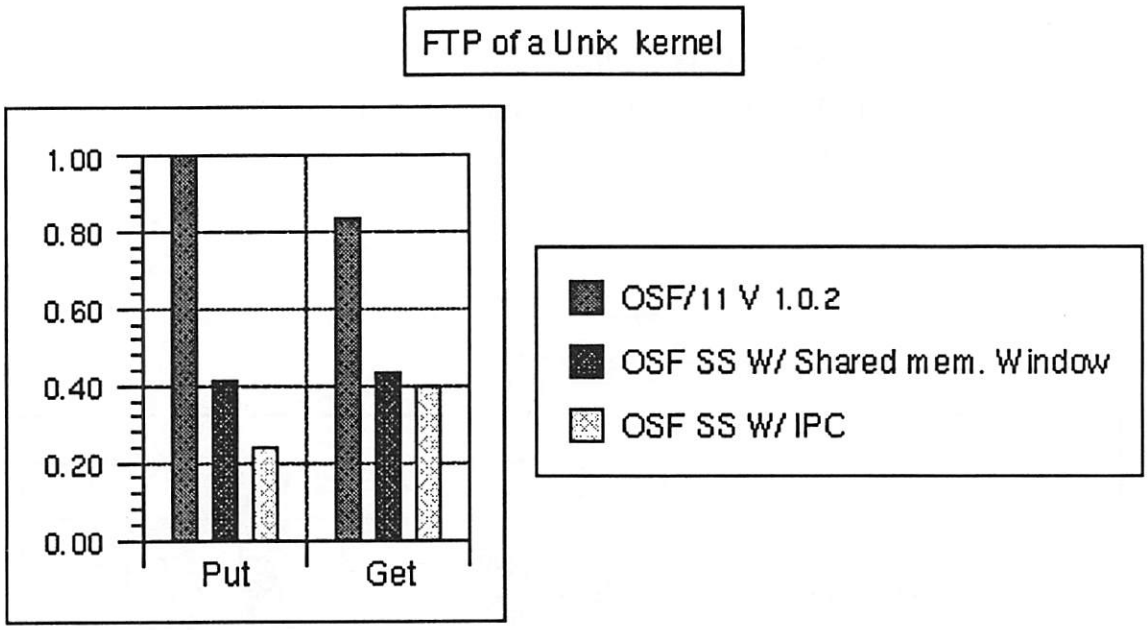




TTCP: Ttcp numbers are much more effective in judging network streaming performance then ftp. This program is purely a network benchmark. As used by us it has no file system interactions. For transmits our tests showed that the OSF single server's performance was less then one third of the integrated system across all three packet sizes. As can be seen in the "TTCP Transmit" bar graph, our modified single server has recovered about half of the difference for all three packet sizes. All three system perform poorly on very small packets, yet still the OSF/1 system is more then twice as fast as the other two systems. Our modified single server was uniformly in the middle of the two systems. We are currently winning back about half of the performance lost by the current interface in transmits. Ttcp receives present a slightly different

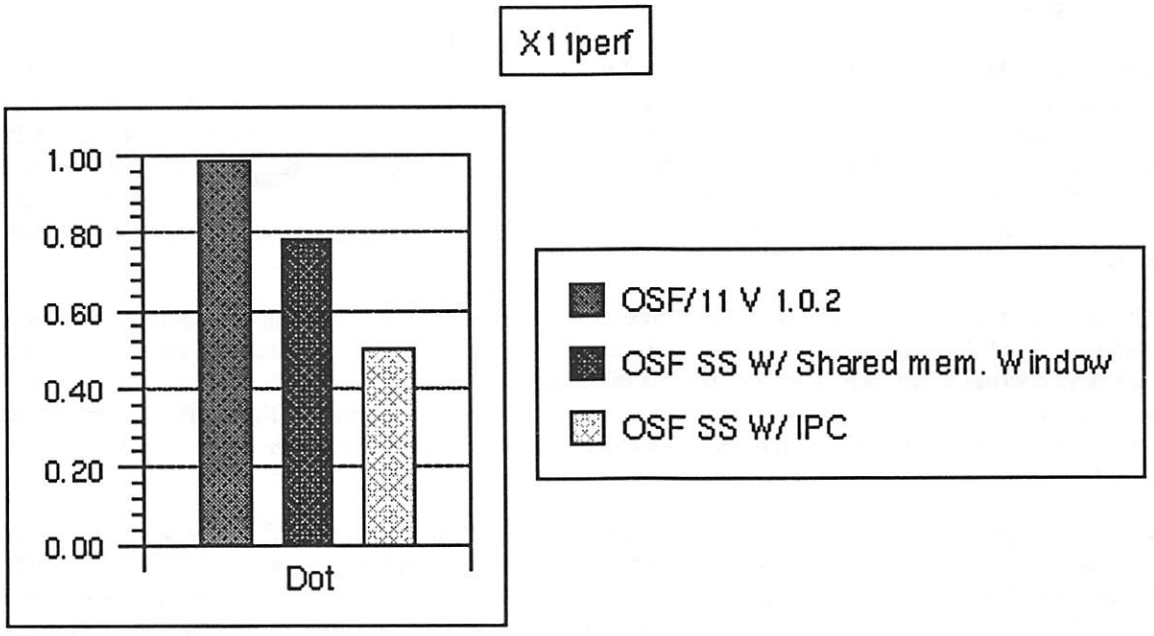
story. On ten byte packets the OSF single server is less then a fifth the performance of the OSF/1 system. Our modified server outperforms OSF/1 by about ten percent. This is the only place where the benchmarks show our server outperforming the OSF/1 system. On one kilobyte packages the OSF single server was about one third of the performance of the OSF/1 system, and we are again in the middle of the two with performance just over half the performance of the OSF/1 system. On 8 kilobyte packages the OSF single server was a little under half as fast as the OSF/1 system, and our modified system was only a little faster then the standard single server.

Ftp was chosen due to the property that it is universally used to check network performance. Although it is not a great network benchmark due to interactions it has with the file system, it is common, easy to use and easy to report. No network performance discussion would be complete without some ftp numbers. This ftp test was the ftp a unix kernel from disk on one machine to /dev/null on the other. The test was run six times in each direction.



FTP: In ftp OSF/1 1.0.2 was the best to both transmitting and receiving files. The OSF single server using the standard IPC based network interface from CMU is about twenty five percent the speed of the OSF/1 system on transmits and just over forty percent the speed of OSF/1 in receives. The closer numbers for receives over transmits are not too hard to believe as the OSF single server's file system, and disk interfaces are not yet final and complete. Work on file system is currently underway in another project. Our enhanced OSF/1 single server suffers from this same problems, and we expect performance of both systems to improve over time. Ftp's reliance on file system performance is one of the reasons why it is not a reliable network benchmark. Our modified OSF single server did provide some performance improvement in the ftp case. In transmitting a file we saw an improvement of almost eighty percent over the standard interface, however we still are only about 45 percent the performance of the OSF/1 integrated kernel. On receives the story is not quite as good. While we are just over 50 percent the speed of the integrated kernel, we are just over ten percent faster then the OSF single server.

Our third and final benchmark is X11perf's dot test. This was run as an X client application from the 386 to an X server running on the NeXT. The benchmark reports back the number of dots per second that it was able to cause the server to plot. This test has the most varied workload of all three tests, not only is the test sending to the network, but we have the intervention of Xlib packaging the server requests up in what it considers to be reasonably sized request.



X11perf: The numbers from the X11perf suite's dot test are valuable because that are not just network streaming number of a single packet size. This test was the closest we came to testing a real network application. It numbers should give us a feel for how the performance of network applications such as X clients would be affected by the network interface. Again here we see that the OSF/1 system performed the best. The OSF/1 single server performed with about half the performance, and our modified server was about seventy five percent the performance of the OSF/1 system.

The results contain few surprises. Most of the benchmarks show that we have made progress towards our goal of closing the gap in performance between traditional integrated kernels and protocol servers. In our benchmarks we see no areas where we are outperformed by the message passing network interface, although there are areas where we have made less progress to date then we had hoped.

In most cases we have reduced the cost of out of kernel networking. In the majority of the streaming tests we have split the difference between the OSF/1 system's performance and that of the standard OSF single server with it's Mach IPC based network interface. There were two surprises for us. In the ttcp receive case of eight kilobyte packages we had a larger drop in performance over the one kilobyte package size then we anticipated. Our working hypothesis is that the buffer pools are being exhausted but we not yet verified this.

The other surprise was in the ttcp receive of ten byte packets we outperformed even the OSF/1 1.0.2 system. Again it is still early in our analysis of the results, however we speculate that this advantage in performance over the integrated kernel stems from reducing the number of scheduling events. Where OSF/1 schedules a thread for each incoming packet, we only need to schedule a thread if the Consumer thread for that queue is idle queue. With lots of small packets we believe that the producer is able to put packets on the queue faster then the consumer can get them off. The consumer just works its way down the incoming queue, not needing to be scheduled for each packet, because it checks for more work before it goes to sleep. The producer being able to tell that the consumer is still working does not need to go to the work of scheduling some consumer to process the packet, as it does in the current OSF/1 system.

While our numbers are not neutral to the OSF/1 system, these numbers from our early prototype give us a belief that we are on the right track. We still are using mechanisms that are very expensive, such as using the device_setstat mechanism to wake the device up to dequeue outbound packets, yet we are still seeing an across the board performance improvement. We are optimistic that our network interface performance will continue to improve.

Discussion:

The disparity between the performance figures of the monolithic kernel and the standard single server give some indication of the costs associated with user space protocol servers. It should be noted that both single servers tested were early prototypes. Their overall performance, as well as their network performance will improve in due course. All of the performance improvement shown by the experimental single server was due to the network enhancements described in the paper. No other modifications to the code base were made. This suggests that as the performance of the standard single server improves so will the experimental server.

The performance numbers suggest that the biggest improvement comes reducing the number of times a buffer is copied. Moving data to or from the device via Mach IPC is expensive. Even the use of IPC to initiate device writes when there is little or no data to be copied is expensive if it has to be done for every network message. Using the shared memory queues reduces the number of copies. But on the system tested, which has a single CPU and a dumb network device, the number of IPC messages is not significantly reduced.

On the test system, writes are almost always synchronous even though the mechanisms are in place for asynchronous, parallel operation. There is only one CPU. After on out-bound packet is queued on an OUT queue the device is started via a trap to the kernel. The kernel dequeues the packet, writes it to the device and looks for something to do. There is never anything to do because the CPU has been busy doing the device write. Since there is nothing to do the routine becomes idle. Eventually the server gets the CPU back and puts another buffer on the queue and checks to see if the device is idle. It almost always is and the cycle repeats. A multiprocessor system or a system with a smart device controller would permit the protocol to enqueue buffers and the device to write them out at the same time.

IN queues are structured in a way that permits the asynchronous nature of receives to be exploited. However, single threaded applications that exhibit request reply behavior work against asynchronous receives. If the protocol server must ACK before the next message is sent then the asynchronous nature of the receive will never be exercised.

Future Work:

In the immediate future we plan to port the system to coherent shared memory and message passing multi-CPU computers. Our system has enough characteristics in common with UPRC [7] that we expect similar performance on a shared memory machine. URPC may suggest to our design. We also plan to port to a platform with a different CPU and I/O architecture. This will give us the experience we need to judge how successful we were at designing abstractions capable of exploiting friendly hardware. Eventually we plan to run other network applications and benchmarks in an effort to refine our understanding of the behavior characteristics of the system.

We plan to continue work to improve the performance of user space protocol server. An investigation into ways to further reduce context switching and scheduling interactions. One approach is to improve the asynchronous operation of device write without the need for the server to explicitly interact with the kernel. For example, some devices can generate an interrupt upon completion of a write or after a timer elapses. This would be a good cheap way to get into the kernel without an IPC message or system call. Another opportunity is when a message is received. At that time an interrupt is generated. Once the received message in enqueued on the correct IN queue the driver could look for outgoing buffers on the OUT queue. Unfortunately the window between the time when the protocol has enqueued something and when decides to call device_start() is very small.

There may be a way for the application C-thread technology to make the window larger by delaying the write. It may also be possible to implement a delayed or gang write policy. Delaying the device_write after the enqueue operation not only allows other enqueue operations to piggy back but it widens the window exploitable by the receive interrupt handler described above.

C-Threads may have an application in the receive path as well. In the protocol server we associate a single Mach thread with each IN queue. This limits that amount of parallelism available to any particular protocol family. If the thread for a particular family blocks the device driver must unblock that particular thread. This approach simplified the design and initial implementation of the prototype. However the judicious use of C-Threads may improve the potential for parallelism and reduce the number of IPC messages. Network threads would not be attached to particular protocol queues. Instead, Device events would wakeup one of several network threads which would look in shared memory for an indication of which IN queue it should start with. When there was no more work to be done with the first IN queue the thread would check all the other queues before blocking.

Summary:

The primary goals of this project were to improve the performance and efficiency of protocol serves using portable technology. A secondary goal was to maximize the reuse of the OSF/1 monolithic kernel code base. The preliminary performance results from the first prototype are encouraging. The current prototype provides enough improvement in performance to continue the project. The perturbation of the code base and the additional complexity needed to manage the shared memory data structures was greater than originally expected.

As has been observed by other researchers [8] current practice in hardware design is not in step with the evolving practice in OS design. The lack of compare-and-swap instructions to build synchronizers, the lack of good segmented architectures or equivalently flexible VM architectures, the costs incurred by PIC code, the costs of context switching or system calls, dumb device controllers, and the lack of support for safe access to devices by unprivileged processes contributes to the difficulty of building safe and efficient servers. To avoid exacerbating the situation operating system designers and CPU designers need to play a greater role in one another's work in the years to come.

Acknowledgments:

The authors would like to thank David Black of OSF and Alessandro Forin of CMU for their contributions to the design and implementation of the system. We would also like to thank Keith Loepere and Paul Neves for their help in designing the various synchronization algorithms.

References:

[1] "Unix as an Application Program", David Golub, Randall Dean, Alessandro Forin Richard Rashid, Usenix Conference Proceedings, Summer 1990

[2] "The Packet Filter: An Efficient Mechanism for User-level Network Code", Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta, Proceedings of the 11th Symposium on Operating Systems Principles, ACM SIGOPS, November 1987

[3] "Locking and Reference Counting in the Mach Kernel", David L. Black, Avadis Tevanian, Jr. , David B. Golub, Michael W. Young, 1991 International Conference on Parallel Processing

[4] "Impossibility and Universality Results for Wait-Free Synchronization" Maurice P. Herlihy, Proceedings of the Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, August 1988

[5] Algorithms for Mutual Exclusion, M. Raynal, Scientific Computaion Series, MIT Press, 1986

[6] Operating System Concepts (chapter5 section2), A. Silberschatz, J.Peterson, P. Galvin, Addison-Wesley, December 1990

[7] "User-Level Interprocess Communication for Shared Memory Multiprocessors", Brian Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy, ACM Transactions on Computer Systems, May 1991, Volume 9 Number 2

[8] "The Interaction of Architecture and Operating System Design", Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska, Proceeding of the ACM Fourth International Conference on Archetectural Support for Programming Languages and Operating Systems, April 1991

An I/O System for Mach 3.0

Alessandro Forin
David Golub
Brian Bershad

{af,dbg,bershad}@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue Pittsburgh, PA 15213

Abstract

The Mach 3.0 I/O system represents a radical departure from its predecessor – Mach 2.5, which relied on the BSD Unix model of device management. The I/O interface in Mach 3.0 supports device drivers that are largely *device-independent*, implemented at user-level, and location-independent. Our approach to device management significantly reduces the size of the kernel's machine-dependent code, enables us to reduce the length of the I/O path, and permits us to transparently manage remote devices on non-shared memory multiprocessor architectures such as the Hypercube. This paper describes the structure and performance of Mach's I/O system.

1. Introduction

This paper describes the design of the I/O system for the Mach 3.0 kernel [Rashid et al. 89]. Mach's I/O system is novel in several respects. First, it supports the notion of "device independent" device drivers. The I/O system separates out generic driver code common to a class of devices such as a screen, an Ethernet controller, or a disk, from code which is only dependent on the device controller *chip* itself, and from the code which is specific to a given processor architecture. Second, the Mach I/O system supports user-level device management of mapped devices, enabling application programs, such as an operating system server, to directly control device activity. Finally, the Mach kernel provides for location-transparent device management which can be accessed through Mach's interprocessor communication (IPC) facilities.

1.1. Device Management for Small-Kernel Operating Systems

Mach 3.0 is an operating system kernel which is intended to be freely portable across a large number of processor architectures, offer network transparency, support efficiently a variety of operating systems implemented as user-level applications, and provide a scheduling interface suitable for the needs of concurrent, Real-Time and parallel programs.

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDS972-90-C-0035.

Previous versions of the Mach 3.0 I/O system made it difficult for us to meet these goals. We describe why this was so, and how the new I/O architecture addresses the problems, in the rest of the introduction.

A Smaller Kernel

The Mach 3.0 kernel had originally inherited the I/O management structure of Mach 2.5, which in turn derived its I/O system from BSD UNIX. Under BSD, devices could only be described as character-oriented or block-oriented devices. This gross characterization made it difficult to share code across functionally equivalent devices which happened to exist on different system platforms. The I/O system simply had no structure to allow similar devices, for example, a monochrome and a color display, to share code, even though the functions provided by the similar devices are nearly identical. We tried, whenever possible, to use the vendor-supplied device drivers when porting Mach to a new system architecture. While this could sometimes reduce the time to port, it resulted in a large amount of nearly duplicated code within the kernel because one vendor's interpretation of how best to drive a device differed from another. Since the drivers were not just device, but also processor dependent, we had no easy way to exploit the similarities. Worse, the drivers from the vendors themselves were nearly always "cloned" from a pre-existing version that handled a similar device, resulting in even more code duplication.

Device-independent device drivers decrease the amount of Mach's machine-dependent code, thereby decreasing the size of the operating system and the amount of time required to port the system to a new system architecture. The generic structure of the new Mach device drivers allows us to wean ourselves from the vendor-supplied drivers, thereby reducing the amount of vendor-owned code in our system. Moreover, maintenance of the system is greatly simplified because changes to the I/O system, for example to increase performance, need only be applied to each class of device, not to each device for each processor architecture for each system.

Real-Time

A second problem with earlier versions of the I/O system was that it was designed to run entirely in kernel mode, generally at high priority in ignorance of other scheduling requirements. This ignorance was because the drivers were often supplied by vendors in the context of a bundled UNIX kernel, which itself has no support for Real-Time computing. The Mach kernel, though, does provide for Real-Time support, so it is necessary to bound the amount of time spent within kernel interrupt handlers down to the "negligible" range.

User-level device management reduces the amount of code that runs in privileged kernel mode, and increases the predictability of the kernel's scheduling algorithms.

High-Performance I/O Devices

While I/O systems of the last 20 years have been measured in megabits per second, those of the next decade are likely to have data transfer rates on the order of gigabits per second. High-speed networks, which can provide data at this rate, and multimedia applications, which can similarly consume data, are two obvious forcing functions pushing on software architectures to support high-performance I/O.

The in-kernel drivers in previous versions of Mach acted not only as device controllers, but also as data buffers. This meant that data would be ferried across the user/kernel boundary as it passed

between the application and the device. For low-performance I/O devices, such as SCSI disks, the additional data transfer time was not important. But, for high-bandwidth applications, it was necessary to reduce the length of the I/O path.

User-level drivers can provide for increased I/O performance because it is possible to avoid expensive data copies (either physical or virtual) between user and kernel space. Data can instead flow directly between the application and the device. This can be done by *mapping* the device directly into an application's address space, just as is done with display devices for graphics-based workstations. Where architecturally possible, the Mach 3.0 kernel allows an application to map a hardware device into its address space.

Location Transparency

While being able to access devices from user-level solves one set of problems, it has the potential to introduce another. It is important to us that we be able to control a device from any machine, not just from the one to which the device is attached. For this reason, we have designed an IPC-based device interface, rather than one based on kernel traps, allowing us to implement remote device driver management. As is customary in message-passing kernels, the device is viewed as a server to which client programs make remote procedure calls (RPC). This ability is critical on "NORMA" (No Remote Memory Access) multiprocessors, in which each processor runs its own instance of the Mach kernel [Barrera 92]. For example, on the Hypercube, we can run the UNIX server on a fast i860 processor and have it drive a SCSI controller attached to a slower i386 processor.

1.2. Some Problems We Weren't Trying To Solve

While designing the I/O interface for Mach 3.0 another thing was quite clear: the user of such interface was not going to be a final user application program, but rather an operating system server such as a filesystem or protocol server. Therefore we have made no attempt to masquerade I/O devices as files or any other "uniform" programming abstraction on the basis that i.) such a uniformity, if desired, should be provided at a higher level, and ii.) providing it at the device level creates unnecessary problems for the many different types of servers that would use the interface. In particular, the Mach kernel is intended to support a variety of different operating system environments, such as BSD UNIX [Golub et al. 90], MS-DOS [Rashid et al. 91], and MacOS, each one exporting its own device abstraction. Our approach allows servers for these operating systems to implement their abstractions at the lowest possible level.

The rest of this paper is structured as follows. In Section 2 we describe the structure of our device independent device drivers. In Section 3 we discuss user-level device management. In Section 4 we briefly describe the IPC-based I/O interface. In Section 5 we discuss some crucial aspects of the performance of the new I/O system. In Section 7 we briefly discuss some related work. Finally, we discuss the system's current status.

2. Device Independent Device Drivers

Early versions of Mach have used pre-existing BSD UNIX device drivers with minor modifications which adapt them to the Mach's virtual memory and thread management systems. These drivers typically come from the machine's vendor, and are therefore different across different vendors. Nevertheless, there is much in common both across hardware devices and across the software that drives

them.

The basic observation that led us in the design of a new implementation paradigm is that hardware devices, especially in the case of workstations, are built with off-the-shelf components, such as video RAMDAC chips, serial line UARTs, SCSI controllers, Ethernet controllers and so on. Although each chip behaves differently, within a class of devices all chips basically perform the same set of functions. For example, every video controller chip has the ability to move the cursor, and every serial line controller chip can be instructed to set speed and parity. Identifying those common functions and encapsulating them at the bottom of a class-like hierarchy produces a system where the code for one chip can be easily replaced by the code for a different one. Not only does this enhance portability, but it allows us to easily integrate new and better versions of chips as they become available over time.

By creating classifications for devices, and then identifying the chip-dependent interface for each class, we are able to write device drivers that are largely independent of the actual make and model of the piece of hardware that they are driving. Instead of character and block devices we now have a set of functionally grouped device drivers including the screen, console, disk, tape, serial line, and Ethernet. Each driver has one or more layers of chip-independent code, which provides both the external interface and implements the logic behind the workings of the device driver itself. Only simple, core functions at the bottom level deal with the hardware directly. Portability and code sharing are greatly increased by this structure. There is no need, for example, to rewrite the code which drives a previously handled SCSI when porting to a new machine that uses a new processor from a new vendor.

Our approach to device independent device management is similar to the one used in Mach's machine-independent VM interface (pmap module) to the various Memory Management Units (MMU) [Rashid et al. 87]. The pmap layer encapsulates MMU dependencies beneath the bulk of the VM system. This greatly increases portability, because only the pmap layer needs to be changed for a new architecture, and code sharing, because much of the pmap layer is constant across similar MMUs.

In the rest of this section, we describe the structure of each of the major device classes supported in Mach 3.0.

2.1. The Serial Line Driver

Serial lines and their drivers are among the oldest components of UNIX's I/O system. The chips in use today, for example, are essentially the same as those that were available ten years ago. Nevertheless, on the software side, each system vendor has supplied its own version of the serial line driver, but they all derive from the original code written at AT&T [Ritchie and Thompson 78]. This situation creates unnecessary code duplication. In addition, we have seen several cases where the same UNIX ioctl is encoded differently because the ioctl interface has semantics that can be specific to device drivers. Since this interface is visible to applications, the proliferation of "similar but not quite the same" device drivers has created binary compatibility problems for us.

The Mach 3.0 serial line driver is split into a device independent component and a small device (chip) dependent part. The device independent component deals with character buffering, and all console-related code including:

- open/close/read/write functions,
- start/stop operations,
- modem controls,

- interrupt handlers for the simple case of devices that work on a one-interrupt-per-character basis.
- send/receive of characters in polling mode to the system console for debugging and error messages,
- switch code for the bitmap driver (mouse and console callouts),
- finding the appropriate console line in a generic kernel,

The chip-dependent layer implements only those operations that manipulate the device registers directly. These include probing for existence, setting of speed, parity and modem control, and moving characters on and off the chip.

2.2. The Screen Driver

The portions of code that are shared across all screen devices are the terminal emulator, fonts, screen saver, interface routines such as open/close/read/write, event handling logic for both motion and keypress events, and other status control operations such as controlling the screen saver and the cursor position.

Code that is specific to each screen device controls probing, notification of open/close operations, character painting at a given location, scrolling, cursor motion, video on/off and blanking, enabling/disabling of vertical-retrace interrupts, and returning the physical address of registers for user space mapping. Even at the lower level, which is chip-specific, we have been able to share some code across devices. For example, all displays based on framebuffers share the same code for painting characters and scrolling text.

Separate from the screen module, but logically part of the same driver are the drivers for the keyboard and mouse. These are structured as devices in their own right, but are only invoked from the serial line or screen drivers and not by general user applications. The keyboard driver remaps the keyboard's keycodes into ascii characters for the terminal emulation task. The mouse driver repacks bytes from the mouse into coordinates and mouse/tablet button keypresses. Device specific components handle the format of the mouse reports and the keycode translation tables.

2.3. The SCSI Driver

Most current workstations provide a single SCSI interface for accessing mass-storage devices such as disks and tapes through a common transport layer.

Our new SCSI driver has three layers. The upper one is specific to each of the major devices defined in the SCSI-2 standard. The code at this layer handles the queueing of requests, tape read errors, bad blocks, disk labels and so on. This layer is implemented as a common source file for open/close/read/write functions and a switch into device-type specific functions for extra open/close/start/restart operations. Common open-time operations include, for instance, dynamically probing a yet-unseen target, and bringing the target online and locking it if it contains removable media. Specific operations for a disk include setting the logical block size, and reading the size and geometry of the disk. Specialized functions, such as disk formatting and bad block scanning, are also exported at this layer.

The second layer defines the encoding of commands into SCSI messages, but also includes other utilities such as a watchdog to recognize a hung SCSI bus, data structure allocation and initialization code, and the definition of the per-target status record.

The bottommost layer handles the hardware proper and only has two interface functions: one to probe, and one to start a SCSI command. There is only one single upcall from this layer, to notify completion of a SCSI command and start the next one for the same device.

A Methodology for Handling SCSI Chips

SCSI chips typically require several interrupts per transaction, therefore it is important to dismiss the interrupts quickly. Some of the older SCSI chips, for example, require between 5 and 21 interrupts per disk read and write operation. We have structured our SCSI chip module as a set of "scripts," which are a list of condition-action pairs. One script might cover all SCSI commands that needs to receive data from the device, another one for transfers in the opposite direction. The condition encodes a possible value from the status registers, and the action is a function pointer. At interrupt time, the status registers are compared against the condition. If they match, the action routine is invoked. Otherwise, control transfers to an error handler associated with the script. At each interrupt, the anticipated condition-action pair is advanced to the next entry in the script until the command completes. For example, disconnections are handled as errors in the processing of a regular, non-disconnecting script. The script pointer is simply saved in the target device's status record and restored later when the target reconnects.

Our use of scripts, which draws on the design of the NCR 53C700, simplifies the writing of the chip-specific code by allowing us to use a single generic control module. Only scripts, action functions, and error handlers need to be written for each new SCSI chip. The more sophisticated SCSI boards, which include a processor, memory and other logic, do not require scripts because they are capable of handling most of the protocol details on their own.

Pushing Harder on SCSI

SCSI is a flexible model for device management — nearly any device can be interfaced via SCSI. In Mach 3.0, we need only write a small amount of machine-independent code to make a new device accessible across all machines. For example, we have connected two machines via a SCSI cable, much like we do with Ethernets, inventing a "host" device that can be used just like an Ethernet. This required only 46 lines of new machine-independent code, and the sharing of another 130 lines with the tape driver. We were able to use the existing structure to handle all of the SCSI nuances. Handling of a CD-ROM only required adding two lines of C code to the existing disk driver to prevent the issuing of write requests. We have dual-mounted the same disk on a DECstation 3100 and a IBM PC with only one additional line of code in the existing DS3100 adapter module.

2.4. The Ethernet Driver

The Mach 3.0 I/O system includes support for only one Ethernet driver based on the Lance chip controller. As most machines use this chip, we have not had much incentive to factor out code common to other Ethernet controller chips. Nevertheless, in importing Ethernet drivers from vendors, we have observed a "cloning" syndrome similar to that for other device drivers (which drive different chips). Our Lance driver is used on four different workstations. The driver copes with a variety of minor system dependencies through the use of callouts to machine-dependent functions that handle the movement of data in and out of the Lance's memory, and for translating a host address into a physical address usable by the Lance chip. Most of the system dependencies are due to the different ways in which the Lance handles DMA across different platforms.

The bulk of the machine independent code in the network driver deals with more general issues

such as allocating and deallocating IPC buffers, delivering messages to users, and using the packet filter [Mogul et al. 87]. This code is common to all Ethernet drivers and can be generalized to any network interface.

3. User-Level Device Management

Devices can be managed from user-level by vectoring all device interrupts out to an application's thread. The kernel maps to user space the device's registers, a shared page containing some control information, and some memory for handling DMA to/from the device. When an interrupt comes, a small interrupt routine[1] saves any volatile register state in the shared page for later use by the user code, and then dismisses the interrupt, typically by disabling the interrupt enable bit in the device, or by reading an "interrupt-acknowledge" register. When the user thread runs it just invokes the driver's interrupt routine as if it were handling the interrupt in kernel-mode. After all necessary processing, the thread then re-enables interrupts in the device.

Our approach to user-level device management allows us to reuse existing kernel-mode drivers to a large extent, even though they run in user-mode. We have generally been able to run a kernel-mode driver in user-mode by providing some simple "scaffolding" for facilities that are normally present in the kernel, such as priority emulation and memory allocation, but not normally present in a user-level application. The only synchronization required for user-level device management is between the kernel's interrupt handler and the application thread. Presently, we use Mach's general `thread_suspend` (from user-mode) and `thread_resume` (from kernel-mode) primitives.

The small interrupt routine that vectors hardware interrupts to threads can be loaded in the kernel either dynamically or statically. Presently, we do it statically at link time, although we could provide a server that does dynamic linking and downloading in kernel space using the system's VM primitives, as is done on the NeXT. The interrupt routine only needs to invoke one kernel function to wake up the interrupt thread.

We should note that our user-level strategy scheme only requires one dedicated thread per device. It does not actually dictate whether this thread runs in user or kernel mode. Indeed on certain architectures, where mapping device registers is not possible, it might be mandatory that the thread runs in privileged mode.

Presently, we are running with user-level drivers for the Ethernet and the SCSI disk. The Ethernet driver was the first user-level driver we wrote, and is in fact the same custom driver (see Section 2) that first ran in the kernel. Our main motivation for moving the driver out of the kernel was our dissatisfaction with the performance of the in-kernel driver. By mapping the driver directly into the UNIX server's address space where the network protocols are implemented, we avoid one extra copy of the data, almost doubling the speed relative to earlier versions of Mach 3.0. In fact, current network performance for throughput intensive applications, such as FTP, is about the same as that for Mach 2.5, which implements UNIX in kernel space.

For the SCSI driver, we initially used the vendor's code directly on a DECstation 5000, and only later moved on to our own device independent machinery described earlier. We did this in order to assess the impact, in terms of performance and programmability, of moving existing, mature drivers out of the kernel would have. Performance is discussed in Section 5. In terms of programming, we were pleased to discover that the effect was minimal, and was all concentrated on the interface between the driver and the scaffolding code, not between the driver and the device. In fact, during our initial port, we didn't try to understand much of the code in the vendor's original driver — it wasn't necessary.

[1] For example, on the MIPS architecture, the routine is 128 bytes.

4. I/O Interface

The I/O interface is defined in a language-independent MiG definition file and consists of the following remote procedure calls:

device_open(*master_device_port*, *mode*, *name*, *device* **)** Open procedure, returns a *device* port

device_close(*device* **)** Close procedure.

device_write(*device*, *mode*, *recnum*, *data*, *num_bytes*, *bytes_written* **)** Write procedure, returns the number of bytes actually written.

device_read(*device*, *mode*, *recnum*, *bytes_wanted*, *data*, *bytes_read* **)** Read procedure, returns the data and the amount of bytes read. Reply can be asynchronous.

device_map(*device*, *protection*, *offset*, *size*, *pager*, *unmap* **)** Map procedure, returns a port *pager* for mapping to user space, usable with **vm_map()**.

device_set_status(*device*, *flavor*, *status* **)** Change the device status, device-specific.

device_get_status(*device*, *flavor*, *status* **)** Inquiry the device status, device-specific.

Device names are strings, and are system-specific. Our convention is to use an alphabetic string followed by an optional decimal number which identifies different instances of similar devices. Record numbers are interpreted in a device-specific manner: a disk uses this unsigned index to point to a physical block, while a serial line just ignores it. Read and write operations can either return data inline or out-of-line. For devices that return data asynchronously, like the Ethernet, for example, a read call can be split in the request and reply sides, possibly with a different thread dequeueing replies. Operations on the status of a device, such as modem control operations on a serial line for instance, are very much device-specific.

Note that any entity that abides by this interface qualifies as a Mach device, whether it is implemented inside or outside of the kernel. The same interface is exported by the kernel for the devices it handles itself, therefore a user application will see no difference whether the driver is implemented in the kernel or in a user process. It is conceivable that a user-space driver could export some other interface, perhaps shared memory based, to other tasks on the same machine. Indeed, the current prototype, in which the SCSI driver is in the same task as default pager [Golub & Draves 91], exports the disk to the UNIX server via the RPC interface and to the default pager via local function calls.

It's important to note that Mach's support for distributed shared memory [Forin et al. 89] does not enable remote mappings of the chip's registers because devices do not access their registers through the memory management unit (MMU).

For devices that are implemented inside the kernel we provide a layer of code that handles VM and scheduling. At this level, we wire pages that are used to move data between user and kernel space. We also use the page-list technique, described in [Barrera 91], to speed up the paths through the VM code.

Scheduling issues are also handled here. Each device-specific function returns a code indicating whether the operation requested was able to complete, or was queued for later processing. If queued, the address of a completion function is noted in the request record. When the request has been handled, the driver's interrupt routine causes the completion function to be executed within the context of a kernel-mode thread. For example, in the case of a device write, the completion function deallocates memory and sends back a simple reply message to the writer with the a completion code indicating success or failure.

5. Performance

We consider two performance measurements for the new I/O system. The first is in terms of the reduction in size of device driver code. This is primarily a function of the new device independent drivers. The second measurement is in terms of performance; that is, how fast can data be pushed through the I/O system.

5.1. Size Considerations

We have observed, on average, a factor of two reduction in the size of device drivers relative to those provided by vendors. Moreover, our new drivers often include additional functionality. The screen driver, for instance, is one fourth the size (MIPS object code) of that shipped by the vendor, and now includes a terminal emulator. The extra code needed to support a hi-resolution screen required only 2KB of object code, compared to over 60KB if the new driver were cloned from an existing one (as is the standard practice). The chip-dependent code in the serial line driver takes about 4KB for each of the two chips we currently support. The support code for the NCR 53C94 SCSI chip is about 10KB, half the size of the vendor's chip-specific code. The most complicated SCSI chip so far needs about 17KB of MIPS object code. The simplest one is about 5KB of Intel 386 object code.

The machine-independent code is also compact. For example, the machine-independent code to support all of the SCSI tapes is about 4KB. The total size of the Lance driver for four machines is less than 8KB. The size of the machine-independent code for the entire I/O system in a generic DECstation configuration is 154KB.

The strictly machine-dependent device code is less than 6KB, and all of that is for handling DMA. Moreover, all of code is written in C. The remaining machine-dependent code in the system is 92KB, including 20KB of debugger support code and 13KB of floating point emulation code. Table 1 summarizes these numbers and shows that the new I/O system is significantly "less" machine-dependent (and therefore more portable) than other components in the system.

DECstation MK64 Generic Kernel

Component	Size (KB)	%
MI I/O	154	96.6
MD I/O	6	3.4
MI other code	364	79.8
MD other code	92	20.2
Total MI	518	84.1
Total MD	98	15.9

Table 1: Maximum Kernel Object Code Sizes.

5.2. Speed Considerations

Our new drivers perform no worse than those that they replace. In some cases, performance is even improved because of the mapped devices and the generic script facilities which allow us to rapidly dismiss anticipated interrupts.

The Screen and Serial Drivers

For the screen and serial drivers, there are no observable performance differences between our new drivers and the vendor's. In the case of the screen driver, this is because the vendor's driver was already mapped into user space, and because the kernel resident code has little impact on performance. In the case of the serial driver, it's because measuring performance differences at the slow speeds of 9600 or 19200 baud (typically the maximum rate for serial lines) is difficult.

The Ethernet Driver

For the Ethernet driver, we measured substantial performance improvements over the vendor's original driver. For example, an FTP using the same 4.3 BSD network code (pre Van Jacobsen) between two DECstation 3100s went from 120KB/sec to 230KB/sec.

The SCSI Driver

Initially, we measured the performance of an out-of-kernel SCSI disk driver which was identical to the vendor's original in-kernel driver. That is, we did not measure the impact that "device independence" had on the performance of the SCSI driver. We discovered that the in-kernel and out-of-kernel drivers performed similarly. The additional cost of having to dispatch a device interrupt out to user-level was insignificant compared to the long seek and rotational delays associated with disks (the average delay we saw was about 10 ms across a number of SCSI disks).

We next measured the impact that our device independent approach had on performance by replacing the vendor's driver (at user-level) with our own. On a DECstation 3100, we saw the maximum disk throughput improve from 700KB/sec to 850KB/sec with our new driver.[2] The throughput here was limited by a slow disk. We then replaced the disk with a faster one, and measured throughput of 1.52MB/sec, which is the maximum rate at which data can be moved between the SCSI buffer and the processor's main memory.

The SCSI driver is a particularly challenging case because of the large number of interrupts required to perform common device functions. We were clearly adding some overhead to the interrupt path. Because many SCSI devices tend to generate many interrupts per hardware operation, we were concerned that extensive coding changes would be required to get good performance. As the performance numbers demonstrate, this turned out not to be the case.

6. Some Observations about I/O Systems

This investigation of the I/O subsystem was originally just motivated by the need of doing a clean, free, reference port of Mach 3.0 to one of the many possible workstations on the market. The findings of the process, and past experiences in porting Mach to the many machines we ported it to are intriguing enough to prompt some more general reflections.

---

[2] In order to measure maximum throughput, we wrote a carefully tuned file-reading program that does reads out of order to maximize "hits" on the sector's location.

6.1. Horror Stories

Cutting the Wrong Corners

Economy is the foremost rule that has driven the design of current workstations. The results are oftentimes detrimental to performance. Most workstation manufacturers choose to include only a a cheap, dumb SCSI chip rather than a smart, more expensive SCSI board. This means anywhere between 5 and 21 interrupts to the CPU per (disconnecting) disk read or write operation. As an extreme case, we have seen an early SCSI disk disconnect on each and every sector transferred. This required 3*16+5=53 interrupts to read an 8KB disk block. We changed our SCSI driver to optionally disable disconnections for selected targets, but such work-arounds should not be necessary.

As another example, we have ported Mach to a multiprocessor which was built without any DMA support for disk I/O. The idea was that a multiprocessor machine can probably waste one processor in dealing exclusively with I/O. The CPU in this case must pick each individual byte out of the SCSI chip, just like a serial line. Unfortunately, the particular SCSI chip chosen would run 5 times faster in synchronous mode — a mode that necessitates a true DMA path to memory.

Balancing Costs

Many customers are willing to pay extra money for faster and color displays. This has generated a variety of solutions and offerings, often concealing important economic and performance considerations. Many users find it hard to understand why a high performance color machine should be slower at scrolling screen text than a monochrome one. At least some of the efforts in designing graphic accelerators, for example, should go into including higher speed screen memory. Moreover, it makes little sense to attach a slow graphics I/O processor to a fast CPU.

Delivering Promised Function

Another area where we hit many obstacles is the one of DMA. Any DMA device that cannot be used to access each and every byte at any physical address creates a software problem which can only be solved by data copies that slow down the machine. This is even more of a problem considering that with today's CPUs, memory is often the bottleneck. We have seen machines that can only DMA two good bytes every other two bytes, some that can only use a good byte every four (and not byte zero), and some that get a good 16 bytes in a row, but only every other 16 bytes. In other instances, the DMA is "normal," but the mapping between physical address and address to be used by the DMA chip is incredibly complicated. DMA chips which can address as much as the CPU can are rare.

Caches

A big cache helps with the performance of user applications, but is less helpful for the operating system [Ousterhout 90]. As for the I/O system, a machine with a DMA chip is essentially a multiprocessor with cache coherency problems which should not be overlooked. If the cache does not snoop the bus, it is necessary to factor into each I/O operation the cost of flushing the cache, which on many machines is not a trivial one, not even for a relatively small address range such as a page size. The cost of flushing can be as high as 25% of the entire page fault cost. In Mach, we can somehow help by avoiding the instruction cache flush for pages that are not mapped (by the user)

with execute permission, as we do on the MIPS architecture for instance. This only mitigates the problem, and only in the case of separated instruction and data caches.

6.2. Suggestions

An I/O system that performs in the gigabyte throughput range will require radical departures from today's practices. High bandwidth will only be possible with large grain data transfers, effective buffering and memory mapping techniques. This is only possible if hardware and software cooperate.

It is important to handle more than one transaction per interrupt because interrupts have a bad effect on cache and CPU (pipeline) performance. High-performance I/O systems will have to reduce the number of interrupts required to handle data transfer. Otherwise, tomorrow's faster CPUs will spend all their time handling one network packet at a time, just as they do today with serial lines. Large data transfers are possible because main memories are large enough to hold data in anticipation of it being used, and modern virtual memory systems are able to effectively cache the data.

Given our experiences with with user mode drivers, we can consider other uses of memory mapping techniques that improve performance. Consider, for instance, a machine where each device is accessible as a separate memory bank on the main memory bus. This large piece of dual-ported memory is where the operating system server allocates buffers used for I/O. This structure gains two advantages. First, the bus is used only for CPU transactions since devices DMA to their local memory. Second, the copy of data in and out of application space is made by using mapped file techniques [Golub et al. 90] only once and in large chunks. This copy eliminates the need for data cache flushes, because the source data can be marked as non-cacheable.

An I/O interface different from that of UNIX would avoid even this one copy. Many other operating systems have successfully used such a buffer "reserve-fill-release" strategy.

An alternative setup is one where the interface between the main CPU and an I/O device is in terms of an external pager interface itself. The device itself is the external pager and interacts with the main CPU in terms of pagein/pageout requests in a fault-driven fashion. This is a generalization of our work on shared memory servers [Forin et al. 89]. The difference now is that instead of only dealing with "communication" issues we also deal with "permanent storage" and data retrieval issues. If the memory mapping is between two hosts, then we have a distributed shared memory semantics. If the mapping is between a host and a peripheral device (disk, tape, printer, scanner), then we will either retrieve data (read fault) from the device or write it to the device (write fault). By mapping, the host communicates to the device the data it wants to address (e.g. what disk blocks). By faulting, the host signifies that the transfer should take place. In this way, we can use lazy evaluation to drive I/O devices. The device now has the advantage of being able to make decisions of its own as to what stays in the main memory and what doesn't. It can, for instance, remove access to a page just because it is convenient to write it out at that particular point in time, or it can prefetch data and supply it to the kernel in anticipation of an upcoming need.

7. Related Work

Other systems use some of the same techniques for I/O management that we have used in Mach 3.0. Jim Gettys [Gettys 91] has recently rewritten the screen driver for Ultrix by factoring the code into chip-specific modules and generic code. In Sprite [Ousterhout et al. 88], device drivers are structured like ours — functionally specialized into much the same set, although the implementation does not stress chip-specificity as much as Mach's. An experimental version of UNIX based on a micro-kernel done at DEC ran with device drivers in user-space [Palmer & Palmer 89].

8. Current Status

The work described here has been a developing part of the Mach 3.0 kernel since the middle of 1989. We invite the interested reader to obtain a copy of Mach 3.0 by way of anonymous FTP to CS.CMU.EDU.

The device independent serial driver has been ported to two chips (DEC DZ7085 and Zilog 5380) on three machines. The screen driver currently handles two monochrome and five color display types, and is used on two workstation types (VAX and MIPS based). Other ports are under way.

The SCSI driver has been ported to four different workstation types (VAX, MIPS, I386, M88k) and handles five different SCSI controllers ranging from the first-generation NCR 5380 to the second-generation NCR 53C94 to the user-friendly Adaptec 1540. Others, outside of CMU, are using these drivers for Mach ports to other systems.

The user-level Ethernet driver has been in use now for almost two years on three versions of the DECstation (2100, 3100 and 5000/200) with a fourth one just completed (5000/120) and a fifth one underway (Omron Luna 88k). It is distributed as part of the single-server UNIX emulator from CMU.

References

[Barrera 91] Barrera, J. S. *A Fast Mach Network IPC Implementation.* In *Proceedings of the Second USENIX Mach Symposium*, This issue, November 1991.

[Barrera 92] Barrera, J. S. *Operating System Support for Multicomputers.* PhD dissertation, School of Computer Science, Carnegie Mellon University, To be completed in 1992.

[Forin et al. 89] Forin, A., Barrera, J., Young, M., and Rashid, R. Design, Implementation and Performance Evaluation of a Distributed Shared Memory Server for Mach. In *1988 Winter Usenix*, January 1989.

[Gettys 91] Gettys, J. E-mail communication posted on the *mach3* mailing list, July 1991.

[Golub & Draves 91] Golub, D. and Draves, R. Moving the Default Memory Manager Out of the Mach Kernel. In *Proceedings of the Second USENIX Mach Symposium*, This issue, November 1991.

[Golub et al. 90] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–95, June 1990.

[Mogul et al. 87] Mogul, J., Rashid, R., Accetta, M. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 39–51, 1987.

[Ousterhout et al. 88] Ousterhout, J., Cherenson, A., Douglis, F. The Sprite Network Operating System In *IEEE Computer, Vol 21-2*, pages 23–26, February 1988.

[Ousterhout 90] Ousterhout, J. Why Aren't Operating Systems Getting Faster As Fast As Hardware? In *Proceedings of the Summer 1990 USENIX Conference*, June 1990.

[Palmer & Palmer 89] Palmer, R., Palmer, L. Informal Communication at the *First OSF Kernel Developers Meeting*, Cambridge, September 1989.

[Rashid et al. 89] Rashid, R., Baron, R., Forin, A., Golub, D., Jones, M., Julin, D., Orr, D., Sanzi, R. Mach: A Foundation for Open Systems. In *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*, page 109–113, September 1989.

[Rashid et al. 87] Rashid, R., Tevanian, Jr., A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, April 1987.

[Rashid et al. 91] Rashid, R., Malan, G., Golub, D., and Baron, R. DOS as a Mach 3.0 Application. In *Proceedings of the Second USENIX Mach Symposium*, This issue, November 1991.

[Ritchie and Thompson 78] Ritchie, D., Thompson, K. The UNIX time-sharing system. In *Bell System Technical Journal*, July 1978.

[Tokuda & Nakajima91] Tokuda, H., Nakajima, T. Evaluation of Real-Time Synchronization in Real-Time Mach. In *Proceedings of the Second USENIX Mach Symposium*, This issue, November 1991.

Moving the Default Memory Manager out of the Mach Kernel

David B. Golub
Richard P. Draves

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, Pennsylvania 15213

(412) 268-7667

Internet: dbg@cs.cmu.edu, rpd@cs.cmu.edu

1. Abstract

We have implemented a default memory manager for the Mach 3.0 kernel that resides entirely in user space. The default memory manager uses a small set of kernel privileges to lock itself into memory, preventing deadlocks against other Mach system services. An extension to the Mach boot sequence loads both the kernel and user program images at system startup time. The resulting system allows the default memory manager to be built and run in a standard user-level environment, but still operates with the high reliability required by the Mach kernel.

The default memory manager is bundled with another component of the Mach 3.0 system: the bootstrap service. This service starts the initial set of system servers that make up a complete operating system based on the Mach 3.0 kernel. Since the real file system may be one of these servers, the bootstrap service needs its own copy of a subset of the file system. This is shared with the default memory manager. Placing these two components outside the kernel allows them to be easily reconfigured with different file systems.

2. Introduction

We have implemented a version of the Mach 3.0 operating system in which all paging traffic to backing storage has been moved outside of the operating system kernel. This is a significant departure not only from previous versions of Mach 3.0, but also from more traditional operating systems, in which memory management for at least temporary memory has been an integral part of the kernel. This work has involved solving a number of technical challenges. The privileges required to manage temporary memory without deadlocking with the rest of the system have traditionally been only available to services embedded within the operating system kernel. We also had concerns about the time and space required to handle paging requests outside the kernel. However, the resulting system runs as reliably as an operating system with conventional temporary memory management. With a reasonable amount of memory, the system runs as fast as previous versions of Mach 3.0.

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDS972-90-C-0035 and in part by the Open Software Foundation (OSF). Draves was supported by a fellowship from the Fannie and John Hertz Foundation.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, OSF, the Fannie and John Hertz Foundation, or the U.S. government.

The default memory manager is an essential component of the Mach 3.0 operating system. It provides backing storage for temporary memory objects. Since its clients may include other components of the system, it cannot depend upon them for any vital services, such as backing storage allocation, without risking deadlock. Traditionally, such vital components of an operating system have been integral parts of the kernel, often using services unavailable to other subsystems. Earlier versions of Mach 3.0 followed suit: the default memory manager was built into the kernel.

Over the past year, we have re-implemented the default memory manager as a task residing entirely in user space. The default memory manager needs only a few kernel primitives to obtain the privileges it needs: wiring down its own code and data, ensuring that data given to it is locked in memory, and priority in allocating new memory. These privileges are part of the normal Mach kernel interface, and potentially available to any user task. All other communication between the kernel and the default memory manager is via the kernel interface used by all other Mach memory managers.

The default memory manager is bundled with the Mach bootstrap service, which loads the initial set of servers. The bootstrap service and the default memory manager live in the same task and share a subset of the file system code to access existing files. Moving this task into user space has required rewriting the Mach kernel boot sequence so that two program images, the kernel and the bootstrap/pager task, can be loaded at system boot time. The kernel and the bootstrap/pager task are built independently. A new program, *makeboot*, combines the two into a bootable system image.

The Mach kernel can now be built without the extra support required for embedded server tasks that were present in earlier versions. There are no more naming conflicts in the kernel between internal kernel routines and user-level interfaces to them. The bootstrap service and default memory manager, similarly, can be built as normal user tasks. The resulting system is as robust as a standard Mach 3.0 system with the default memory manager inside the kernel. Performance on machines with reasonable amounts of memory is essentially identical.

3. In-Kernel Default Memory Management

A manager for temporary virtual memory must operate under some very adverse conditions. In conventional operating systems, the temporary memory manager must be able to allocate storage when no other component of the system can do so. In Mach, the temporary memory manager is, by default, the memory manager of last resort for all other memory managers. This role of *default* memory manager forces it to avoid most standard system services. These constraints have kept the temporary memory manager intimately tied to the kernel in most operating systems, including earlier versions of Mach.

Operating systems have traditionally provided a variety of memory management services implemented within the operating system kernel. [1] [5] They include:

- Allocating temporary memory for user programs and system services.
- Mapping executable files and data files into a task's address space.
- Mapping IO devices such as graphics buffers.
- Sharing files or temporary memory between tasks.

The external memory management interface present in Mach 2.5 and Mach 3.0 has allowed most of these services to be provided by servers outside the kernel. However, temporary memory management has still resided within the kernel.

Temporary memory management has traditionally been implemented in the kernel because it must make use of other system services at times when they would normally be unavailable or risky to use. Temporary memory usually does not have pre-allocated backing storage. The temporary memory manager must allocate the backing storage when it is needed. Since pages are written to backing storage when the system is short of main memory, the routines that allocate backing storage must be able to operate in such conditions. Furthermore, allocating backing storage often requires allocating memory within the memory manager, further increasing the strain on main memory.

In a Mach 3.0 system, the temporary memory manager acquires an even more difficult role: that of default or last-resort memory manager for the entire system [6]. Memory management services may be provided by any task in the system, not only by trusted system components. Therefore memory managers cannot be assumed to be timely or reliable. A memory manager may take an arbitrary amount of time to write a page to its backing storage. It may also malfunction (accidentally or maliciously) and not write the page at all. During the time the page is in transit (paged out to its memory manager, but not yet written), it still consumes main memory. If the operating system is short of memory, this in-transit page must be paged out. Since the page is viewed as temporary memory while it is in transit, the temporary memory manager thus assumes responsibility for it.

The temporary memory manager therefore must not depend on other servers in the system, since these servers may depend on the temporary memory manager themselves. This requirement has, so far, kept the temporary memory manager within the Mach kernel.

4. Constraints on the Default Memory Manager

The default memory manager faces several stringent implementation constraints, because of its crucial role in the Mach system. It cannot depend on any system service that in turn could use it to provide temporary memory. It must also be able to function when the system is short of memory.

- The default memory manager must be resident - there is no other service to page its code and data into memory.
- All pages moving between the kernel and the default memory manager, and between the default memory manager and its backing storage service, must remain resident.
- The default memory manager cannot block to wait for more physical memory, since freeing physical memory requires that its contents be paged out (through the default memory manager).
- If the default memory manager is grouped with another service, the threads that handle default memory manager functions must be distinguished from threads that do not. When the system is short of memory, the unprivileged threads (those not handling default memory manager functions) may block waiting for physical memory. If the default memory manager's threads can block waiting for the unprivileged threads, a deadlock will result.
- The default memory manager cannot block to wait for the file system to allocate temporary disk storage, since the file system itself may be pageable. This requires that temporary paging space be allocated in advance, or that the file system be as privileged as the default memory manager.

To meet these requirements, the Mach 2.5 default memory manager was implemented as a task bound tightly within the kernel. It used several services that were unavailable to other components of the system.

- The default memory manager was allowed to allocate kernel memory even when the system was short of memory. Threads within the default memory manager were marked as *vm_privileged* threads, and were allowed to allocate memory when other threads would block.

- Threads within the default memory manager were marked as unswappable. The kernel could unlock and page out the entire kernel stack of a long-waiting thread when short of memory. Doing this to a default memory manager thread would, of course, deadlock the system.
- The default memory manager used the existing BSD 4.2 file system code to manage its disk storage and allocate new disk space for paging. The code was written to run in a standard kernel environment, which assumed that all of the code and data was resident, and the kernel stack of any thread executing the file system routines was locked into memory.

However, all paging traffic between the kernel and the default memory manager took place through the external memory management interface - an interface designed to be used with servers residing outside the kernel.

The original Mach default memory manager therefore used a mixture of user and kernel services. It moved pages to and from the kernel using the external memory management interface, as if it were a user task. It accessed the kernel memory and disk allocation routines through its privileged position inside the kernel. This structure, preserved even in the first Mach 3.0 implementations, required the kernel to provide the equivalent of a user-mode interface internally. It spoiled the clean separation between the Mach kernel and user-mode functions.

5. Moving Out of the Kernel

To move the default memory manager out of the kernel's address space, the privileged kernel functions had to be replaced by equivalent functions accessible from user programs. Fortunately, the default memory manager already used the standard external memory management interface to handle paging operations, using external data types (ports) to represent memory objects, rather than handles to the kernel's internal data structures.

Only a few kernel operations were accessed directly: allocating disk storage, synchronizing threads, allocating locked-down memory, and preventing threads from being swapped out. These operations were replaced with equivalent user-mode routines, or were added to the kernel interface where missing.

5.1. Kernel Interface Extensions

The kernel interface has been extended with two calls originally proposed [2] for supporting real-time Mach:

-
```
kern_return_t vm_wire(
        priv_host_t     host_port,
        task_t          task,
        vm_address_t    start,
        vm_size_t       size,
        vm_prot_t       access)
```

  Locks ("wires") the specified range of virtual addresses for the task into memory. Accesses denoted by *access* cannot fault. Memory is unlocked by specifying no wired access (*VM_PROT_NONE*).

-
```
kern_return_t thread_wire(
        priv_host_t     host_port,
        thread_t        thread)
```

  Permanently acquires a kernel stack for the thread, so that the thread can always run. Allows

the thread's requests for physical memory to always succeed.

Both of these calls are only accessible to tasks with send rights to the privileged host port. There is currently no other resource control on memory allocated by these calls.

An existing kernel call identifies the default memory manager to the kernel:

```
kern_return_t vm_set_default_memory_manager(
        priv_host_t     host_port,
        mach_port_t     *default_manager)
```

Returns the old value of the default memory manager service port. If the supplied value for the default manager port is not MACH_PORT_NULL, the kernel will use this port as the default memory manager service port.

The kernel uses the default memory manager service port to request the default memory manager to handle paging for temporary objects. It also arranges that memory sent to the task with receive rights for this port remains locked down in physical memory: see section 5.5.

5.2. File System Access

The largest kernel function used directly by the Mach 2.5 default memory manager was the file system code. Since the file system code was resident, the default memory manager could take advantage of it to allocate new paging space. However, this required that the all of the file system code be resident, and that any thread that accessed the file system also be resident while it was executing that code.

To preserve the flexibility of being able to allocate new paging space would have required us to include the entire BSD file system code in the default memory manager. We decided instead to separate the file system and the memory manager, and use other mechanisms to allocate paging storage.

The default memory manager bypasses the disk allocation problem by using a pre-specified list of disk blocks. At system startup time, the bootstrap sequence looks for a paging file with a well known name: */mach_servers/paging_file*. It then reads the index information in the file to obtain the disk blocks occupied by the file, and passes them to the default memory manager. The paging file must, therefore, be pre-allocated with all of the blocks needed for paging.

5.3. Synchronization and Locking

The default memory manager uses the CThreads package [3] to provide locking and synchronization between different threads. The CThreads package provides two synchronization primitives: mutual exclusion locks, for controlling exclusive access to data structures, and condition variables, used by multiple threads to wait for events and signal them. Mutual exclusion locks were used to replace the simple spin locks used inside the kernel. Explicit calls to the kernel sleep and wakeup primitives were replaced with condition variables. The CThreads package does not provide multiple reader/single writer locks, but since these were used only in one routine in the file system, they were easily replaced with exclusive locking.

5.4. Allocating Locked-Down Memory

We replaced the standard C library memory allocator (*malloc* and *free*) with our own routines. These call the kernel routine *vm_allocate* to allocate full pages. This is replaced by a routine that allocates memory from the kernel (using *vm_map*), and locks it into memory using the new *vm_wire* call. The *thread_wire* call allows the threads in the default memory manager to always allocate memory.

5.5. Locking Memory sent to the Default Memory Manager

The default memory manager receives memory in messages from only a few sources in the kernel and the device service. These have been modified to know when they are sending memory to the default memory manager, and to lock it in memory.

- The pageout daemon locks pages being paged-out from temporary memory objects. These memory objects are created by the kernel to hold temporary memory, or to shadow memory that is passed copy-on-write from one task to another. If a page in one of these objects is sent to the default memory manager, it is locked down.
- The device service locks memory being read from backing storage devices (typically the disk) to the default memory manager. The *device_read* routine looks at the task holding receive rights for the destination of its reply message. If that task is also the receiver for the default memory manager's service port, the memory is locked down.

5.6. Locking Memory sent to Backing Storage

The default memory manager writes data to disk via the kernel device service, using the *device_write* call. This routine must not allow the data to be paged out, and must not block to wait for other paged-out data to be paged in. To avoid blocking, the *device_write* request message is processed in the context of the calling thread, not by a separate device service task. The message-send operation does not return until the memory to be written is queued for the disk.

If the device service were a separate task, it would need two classes of service threads. One set would handle write requests from the default memory manager, containing locked-down memory; the other set would handle requests from all other clients, containing pageable memory. If one pool of service threads handled both pageable and locked-down memory, a write request from the default memory manager could be blocked because pageable memory in a previous write request was paged out. This would result in deadlock, since the device service would need the default memory manager to retrieve the paged-out memory.

6. User-mode Environment for the Default Memory Manager

The default memory manager is built and operates as part of the Mach 3.0 bootstrap service task. This task is the first to run in the system once the kernel is initialized. In addition to the default memory manager, it contains the bootstrap loader for user-mode servers. Since both of these components share a common file system, they were moved out of the kernel as a unit.

6.1. Structure of the Bootstrap Service Task

The bootstrap loader loads and executes the first server task: the BSD server for the BSD single-server system, or the configuration manager for the multi-server operating system. It makes use of a bootstrap file system module to find the server file and read it into a new user task. It passes the privileged host port and the device server port to the initial server, thus giving the initial server access to all system privileged operations.

The default memory manager runs after the bootstrap service. It uses the bootstrap file system to find the paging file, and to read and write the file as paging operations take place. It exports an interface to allow other tasks to use it as a shared memory service; however, it provides no network consistency management, so the objects it exports cannot be shared across multiple machines.

The bootstrap file system that both of these components use provides a small subset of the operations available on a BSD 4.2 or 4.3 file system. It can look up files by device and name, and read and write existing data blocks in files. It cannot allocate new blocks to files, or create new files. To do so would conflict with the real file system. The interface to the bootstrap file system is well defined; the code can readily be replaced by one that understands a different disk format.

6.2. Loading Two Boot Images

Since the default memory manager and bootstrap service have been separated from the kernel, there are now two executable images (kernel and bootstrap) to be loaded when the system is run. We decided not to modify the existing initial bootstrap loaders to load both images. Most of them are proprietary to the individual manufacturers; in at least one case, we do not even have access to the source code. Instead, we expanded the kernel boot file to contain both images.

A new program called *makeboot* builds an executable out of both boot images (kernel and bootstrap) so that the complete image of the default memory manager and bootstrap task is in the kernel data segment when the kernel is loaded. The kernel then moves the bootstrap image to a newly created user task and starts a user-mode thread in it.

The boot file looks like a normal bootable image, but has no uninitialized storage (BSS) or symbol table. The size of the combined text and data segments is set to include the entire file. At the start of the kernel uninitialized data (*BSS*) is a *struct boot_info* describing the sizes of the rest of the file:

```
struct boot_info {
    vm_size_t   kern_sym_size;   /* size of kernel symbol table */
    vm_size_t   boot_image_size; /* size of bootstrap image */
    vm_size_t   load_info_size;  /* size of load information for
                                    bootstrap image */
};
```

Following this are three new sections:

- The kernel symbol table. This may include information pulled from the normal executable file header to make the symbol table self-describing.
- The boot image. This is identical to the bootstrap's executable file, except that its symbol table has also been made self-describing.
- Loader information for the boot image. This is in a compiler-independent format, to reduce the amount of machine-dependent code needed in the kernel.

At startup, the Mach boot file is loaded into contiguous physical memory. The kernel must, as its first action, move the symbol table, the bootstrap image, and the load information out of its BSS section. In addition, it must align the bootstrap image on a page boundary, so that the bootstrap image can be mapped to the bootstrap task rather than being copied. All of this may have to be done before virtual memory mapping is set up, since the kernel typically uses the first few pages of physical memory after the BSS for building page tables.

The kernel proceeds with its normal initialization. After starting all the internal threads, it creates a task and thread to run the bootstrap code. It then creates a memory object mapping the resident pages holding the bootstrap code. It maps this memory object into the first task as it would a normal executable file. It then allocates a stack for the task, passes it a small set of arguments, and starts it running in user space.

7. Results

Moving the default memory manager outside the kernel has both benefits and drawbacks. The build environment for both the kernel and the default pager (and bootstrap task) is considerably simplified, since user and kernel environments are not mixed. The system operates almost as fast as previous versions of Mach 3.0 that contained the default memory manager within the kernel. However, it does occupies more physical memory than previous versions. Using a pre-allocated paging file is adequate for development systems, but not for production use.

7.1. Building the System

Building the Mach 3.0 kernel and default memory manager is much easier. Both the kernel and the default memory manager are now self-contained. A new bootstrap service can be added to the Mach 3.0 boot file without rebuilding the kernel.

The kernel no longer has to support a user environment for otherwise self-contained servers. There is no longer a naming conflict between kernel interface routines called by servers and the kernel functions themselves. The default memory manager no longer has to distinguish global kernel names for ports from its own local names for them; this was a source of confusion in the Mach 2.5 default memory manager.

The default pager and bootstrap service can be built as a standard user task, using the Mach kernel primitives and device support routines, and the standard CThreads library. Where we have re-implemented routines from the standard Mach libraries, we have done so for performance or space reasons, not for new functions.

The kernel and the bootstrap service are built independently and combined with the *makeboot* program. If someone wants to experiment with a new default memory manager, or change the bootstrap file system to accommodate different disk formats, it is not necessary to rebuild the kernel. Only the boot file needs to be rebuilt.

7.2. Performance

Performance figures, comparing the MK63 release of Mach 3.0 with the out-of-kernel default memory manager, are shown in table 7-1. All measurements were run on a 25Mhz 386 processor with 8 Megabytes of memory and an ISA bus.

When operating with a reasonable amount of main memory, moving the default memory manager outside of the kernel has only a small effect on overall system performance. A memory-intensive paging benchmark, paging 7 megabytes of virtual memory against 6.5 megabytes of available physical memory, runs less than 1% slower with the default memory manager out of the kernel. We expect that a balanced workload, not dominated by paging operations, will show no difference.

We expected that overall system performance would be unchanged by having the default memory manager outside of the kernel. When the system is paging heavily, disk operations would dominate the

Memory Size, Megabytes		*Elapsed Time, Seconds*		
Maximum Memory	*Available Memory*	*MK63*	*Out-of-Kernel*	*Difference, Percent*
8	6.5	97.0	97.6	0.6
4	2.5	124.2	131.4	5.8
3	1.5	131.6	138.9	5.5

Table 7-1: Performance of out-of-kernel system vs. MK63

cost anyway. Otherwise, the only difference would be the cost of crossing the user/kernel boundary. The previous implementation of the default memory manager was already structured as a separate task, communicating with the kernel and the disk driver via IPC: this already forced messages to be copied and data to be remapped from the default memory manager's address space to the kernel's address space. The only added cost would be the actual kernel-to-user mode switch. These expectations correspond to what we have measured.

As the available memory decreases, the performance drops, but slowly. With 2.5 megabytes of memory, the paging test runs about 6% slower. The extra memory taken by the out-of-kernel bootstrap task does not account for the slowdown: adjusting the available memory to compensate does not change the performance figures.

One cause of the slowdown is the difference between the kernel and the CThread locking facilities. Mutual exclusion locks in the kernel compile into null routines when the kernel is built for a uniprocessor; since kernel threads are not preemptible, locks are not needed when the thread cannot block. Locking is always needed, however, when running in user space, since user-mode threads are preemptible. When the default memory manager was first moved outside the kernel, it grabbed and released a mutual exclusion lock nine times while reading one page from the disk. Reducing the number of lock/unlock pairs to four reduced the performance degradation from 10% to 6%. We expect that further tuning of the code will regain nearly all of the lost time.

7.3. Memory Usage

The sizes for the kernel and bootstrap image are compared with the size of an MK63 kernel in table 7-2. The extra memory used by the out-of-kernel default pager system is broken down in table 7-3. The loaded image occupies more space than it does in the file because the text and data segments and the symbol tables are rounded to page boundaries (4096 bytes).

File	*Text*	*Data*	*BSS*	*Symbols*
MK63	315360	33096	54364	69524
Kernel	294880	32224	52992	65386
Bootstrap	40928	2676	3616	19345

Table 7-2: Executable Image Sizes

Splitting the default memory manager out of the kernel does increase the system's locked-down memory usage. The default memory manager's code and data is still locked in memory. The C Threads library is

Size in KBytes	*Used By*
48	Code and Data for Bootstrap
24	Symbol Table for Bootstrap
-24	(Code and Data Removed from Kernel)
-4	(Symbol Table Removed from Kernel)
12	Page Table for Bootstrap Task
40	CThread Stacks for Each CThread
4	Extra Waiting Stack allocated by CThreads
12	Temporary Storage Managed by Memory Allocator
112	Total

Table 7-3: Extra Memory Used by Bootstrap Task

larger than the equivalent kernel locking primitives, and is not shared with other functions as the kernel primitives are. Each thread in the default memory manager has a user-mode stack as well as a kernel-mode stack. Since the threads must be wired, their kernel stacks cannot be discarded or handed-off to another thread.

Careful work on the default memory manager could reduce the extra memory usage, but not eliminate it entirely. The user-mode stacks could be reduced to 4 K bytes each, since only the bootstrap server loading code needs as much as 8 K bytes of stack. It would also be possible to discard the code pages occupied only by the bootstrap service loader routines once they have been used. In a production system, the kernel and bootstrap loader would be configured without run-time debugging information: this would save 20 K bytes from the bootstrap task alone.

7.4. Backing Storage Allocation

Using a pre-allocated paging file is currently the least satisfactory aspect of the default memory manager design. Mach 2.5, since it integrated the file system with the default memory manager, could use a variety of paging space allocation strategies: reserved disk partitions, pre-allocated paging files, and expandable paging files. In Mach 3.0, the file system is expected to be separated from the default memory manager. The two tasks must cooperate to supply more disk blocks for paging; otherwise they could both allocate the same blocks from the file system, destroying it. There is currently no way to give more paging space to the default memory manager; one could be added.

8. Further Work

Since the bootstrap service is now independent of the kernel environment, we can experiment with adding additional functions. The default memory manager interface could be extended, as previously mentioned, to request additional paging storage from the file system and to return paging storage that is no longer needed. The bootstrap loader could be used to load all of the servers for the multi-server BSD emulation, not just its configuration service. The bootstrap task is also the logical place to put any out-of-kernel disk drivers [4]that need to be shared between the default memory manager and the file system. To accommodate the space taken by these functions, we would need to restructure the bootstrap service so that code used only at startup can be freed.

For the adventurous, the default memory manager could even be made into a loadable server, separate from the bootstrap service. The default memory manager views backing storage as a list of disk blocks, not as files. Therefore it does not actually need to share the file system code with the bootstrap service. The default memory manager could be loaded and run just as any other system server; at some later time (for example, when *swapon* is executed), a file system server could provide it with a list of disk blocks to use.

9. Conclusions

We have shown that moving all memory management outside the Mach 3.0 kernel is not only possible, but beneficial. The resulting system runs as reliably and almost as fast as earlier versions of Mach 3.0. It is considerably easier to build and can more flexibly be reconfigured. These benefits are obtained at the cost of only a small amount of extra memory.

References

[1] L. A. Belady, R. P. Parmelee, and C. A. Scalzi.
The IBM History of Memory Management Technology.
IBM Journal of Research and Development 25(5):491-503, September, 1981.

[2] David L. Black.
Mach Interface Proposals - Priorities, Handoff, Wiring.
Internal Mach Project Memo - 13 August 1989.

[3] Eric C. Cooper and Richard P. Draves.
C Threads.
Technical Report, Department of Computer Science, Carnegie Mellon University, July, 1987.

[4] Alessandro Forin, David Golub, and Brian Bershad.
An I/O System for Mach 3.0.
In *Proceedings of the Second Mach Symposium.* The UseNIX Association, November, 1991.

[5] E. L. Organick.
The Multics System: An Examination of its Structure.
MIT Press, Cambridge, Mass., 1972.

[6] Michael W. Young.
Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.
PhD thesis, Carnegie Mellon University, November, 1989.

User-Level Physical Memory Management for Mach

Stuart Sechrest
Yoonho Park

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Michigan

Abstract

We have developed an extended version of Mach 3.0 that allows physical memory managers to run as user-level processes, and which allows the memory requirements of these managers to be balanced. Physical memory is a resource for which there are a number of potential competitors whose diverse uses of physical memory may require diverse management policies. Flexibility in physical memory management policy is important to database managers, multimedia file systems and persistent object storage managers. Our architecture allows the control of physical memory page frames to be assigned to processes running outside the kernel, with page frame records shared between the kernel and these managers. Control of page frames can be reassigned among physical memory managers by a balance manager.

1 Motivation

Physical memory is a resource for which there are a number of potential competitors. RAM is used not only as as backing store for processes' virtual address spaces, but for caches maintained by file systems, persistent object stores, and database systems, as well. The diverse uses of physical memory may require diverse management policies. Flexibility in physical memory management has been a concern of database researchers [11, 1] for a number of years. We believe it will become a matter of increasing concern in multimedia file systems and persistent object storage. As Stonebraker [12] points out, buffer managers may be able to take advantage of semantic knowledge about the data to make more effective use of physical memory. An image data manager can, for example, retain images at reduced resolution, rather than flushing them entirely.

We believe that a physical memory management system must balance the requirements of these competitors for memory, while allowing effective management policies to be implemented. Mach 3.0 does not address this need. We have therefore developed an extended Mach 3.0 system that allows multiple physical memory

managers to run as user-level processes, and allows the amounts of physical memory assigned to these managers to be balanced dynamically.

Mach 3.0 maintains a list of free pages selected by a version of the global clock page-replacement algorithm. This algorithm seeks to locate and free the least recently referenced (global LRU) pages. This algorithm is widely used, but may make inappropriate selections for particular types of data. It has been suggested, for example, that for joins in relational database queries, one may want to replace the *most* recently used page within a relation [1]. The division of physical memory into pools can be used to protect one process from the paging of another, as in the VAX/VMS operating system [4]. Our intention, however, is to use pool boundaries to protect *subsystems* from one another and alter these boundaries through explicit balancing operations. This is particularly important when the cost of storing data to and retrieving data from permanent storage can differ significantly.

2 Related Work

Current systems generally have limited interfaces allowing a user to affect page replacement choices. Mach 3.0 provides a call that allows certain pages to be pinned in memory. SunOS 4.1 provides the call *madvise()*, allowing applications to advise the kernel on the likely pattern of accesses in a region of the application's address space [7]. The application can state that access will be random or sequential, or that certain pages should be held and others released. The kernel can take these claims into consideration while implementing page replacement.

The QuickSilver operating system [3] placed important physical memory management responsibilities outside the kernel [14]. The QuickSilver kernel is extremely small. Management of physical memory is therefore entrusted to a server process. The page tables are shared between the kernel's and the server's address spaces. To implement page replacement the server invokes a kernel call which locates the (approximately) least recently used pages using a clock algorithm. Thus, only one page replacement policy is supported.

PREMO [6], an extension of Mach 2.5, allows a user-level program to implement its own page replacement policy by extending the external pager interface. The external pager is consulted when one of its pages is to be placed on the free list, and is allowed to substitute an alternative page. While this approach allows the implementation of alternative replacement policies within a pool of pages controlled by an external pager, the selection of the pool from which the victim is to be drawn is still based on a global LRU strategy. The PREMO approach does not take into account the differences between pools and may allow some pools to be unfairly victimized. Sprite [8] began to address the problem of balancing competing claims on physical memory by dividing memory between backing store for virtual

address spaces and file system buffer cache, but dynamically adjusting this division. Performance was improved, provided that the virtual memory system was given preference over the file system. The Sprite approach, however, was limited to two pools, and it did not allow the implementation of new replacement policies in any simple way.

3 Architecture

3.1 Physical memory management in Mach 3.0

In Mach 3.0 the majority of the physical memory of the system forms a single paging pool used as a cache for memory objects [5]. Page frames are represented within the kernel at two levels of abstraction. Machine-dependent details of the host memory architecture are hidden beneath an idealized *pmap* interface. Machine-independent records for page frames are kept in the resident page table. The kernel code responsible for handling page faults and for managing physical memory manipulate the resident page table information and make calls through the pmap interface.

The system virtual memory cache is managed by the kernel page-out daemon using an approximate LRU algorithm. With the exception of the *vm_pageable()* call, a privileged call that pins pages in memory, user-level programs have no control over this algorithm. The system page-out daemon implements a global clock algorithm. The available page frames are placed on an active, an inactive, or a free list. The page-out daemon seeks to maintain the list of free page frames, by moving page frames that have not been recently used from the active to the inactive list and from the inactive to the free list. In doing so it relies on calls through the pmap interface to access hardware maintained access information.

In addition to ordinary paging, page frames can be removed from the free list for use as storage for kernel structures [9]. These pages are freed by placing them on the active list, where the page-out daemon will eventually determine that they can be freed.

3.2 User-level PODs

We have developed an extended version of Mach 3.0 that divides the physical memory into a number of pools, managed separately by processes called PODs (for Page-Out Daemons). Because we want to explore physical memory management policies for new areas such as multimedia file and database management systems, we want to simplify the creation and configuration of systems implementing diverse physical memory management policies. PODs therefore run as (trusted) user-level programs, sharing access to kernel data structures (see Figure 1). As noted above,

the resident page table in the Mach 3.0 kernel is threaded with lists maintained by the kernel's page-out daemon. We allow the page table to be mapped into the address spaces of user-level PODs. PODs can also run as part of the kernel, although kernel PODs are not required. A *balance manager* running in the kernel shifts page frames between the pools maintained by the various PODs.

Page faults on memory objects are handled by the kernel. Every memory object is assigned to a POD, and its page faults will be satisfied from this POD's page frame pool. It is the responsibility of the POD to maintain a list of free page frames within this pool. Figure 2 illustrates the relationship of memory cache objects, POD records and the resident page table. The pool of page frames associated with a POD need not (and in general will not) be contiguous. Instead the records for the frames in a POD's pool appear on one of two lists whose heads are in the POD record. The *in-use list* contains page frames known by the POD to be in use, while the *free list* contains the remainder of the pool's frames.

A page frame on the free list is either *free* or *allocated.* (Figure 3 shows the state transition diagram for a page frame.) The kernel allocates pages from a POD's free list. The new state of these page frames is recorded in the resident page table. These pages, however, are not moved automatically to the in-use list. For short-lived objects, these page frames may quickly be freed again. Their state recorded in the resident page table will revert to free.

The state of a page frame on the in-use list is either *in-use* or *must-free.* When the POD is awakened, allocated page frames are removed from the free list and placed on a transfer list. This list is passed (by reference) to the POD, which incorporates them on its in-use list. Their recorded state is changed to in-use. The kernel may deallocate in-use page frames if, for example, an object terminates. In this case they are left on the in-use list, but their state is changed to must-free. Must-free pages and page frames deallocated by the POD are removed from the in-use by the POD and placed on a transfer list. This "laundry list" is passed (by reference) to the kernel.

The POD is awoken by the kernel both periodically and when its free-list falls below an agreed upon low-water mark. In the extreme case, the POD can set the low-water mark to empty and, thus, arrange to be called for every page frame assignment. Ordinarily, however, page frame assignments can be made from the free list without requiring a call to the POD. It is the POD's responsibility to ensure that the free-list size exceeds the low-water mark. When invoked the POD returns to the kernel a list of page frames to flush along with any must-free page frames. The kernel must remove the flushed pages from the memory objects to which they belong and, possibly, invoke external pagers to save modifications. The policies and data structures used to determine which page frames to flush are entirely the concern of the POD.

The architecture outlined here has certain assumptions that should be stated ex-

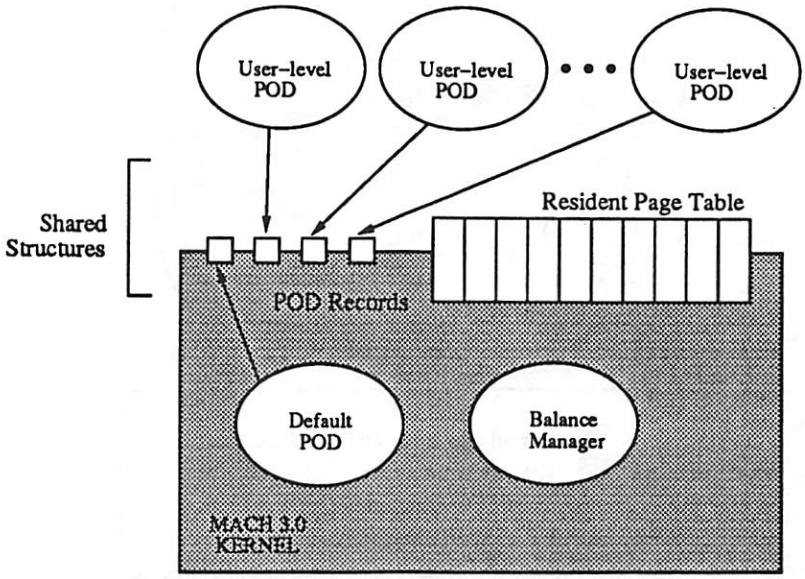


Figure 1: User-level physical memory managers (PODs) share access to kernel data structures

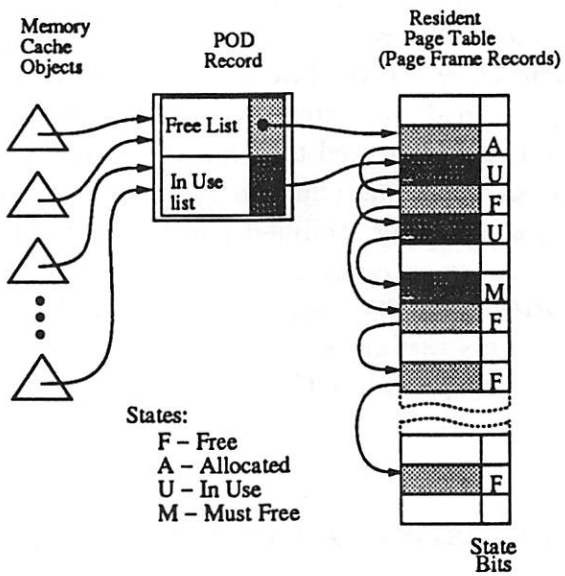


Figure 2: Memory architecture for user-level physical memory management

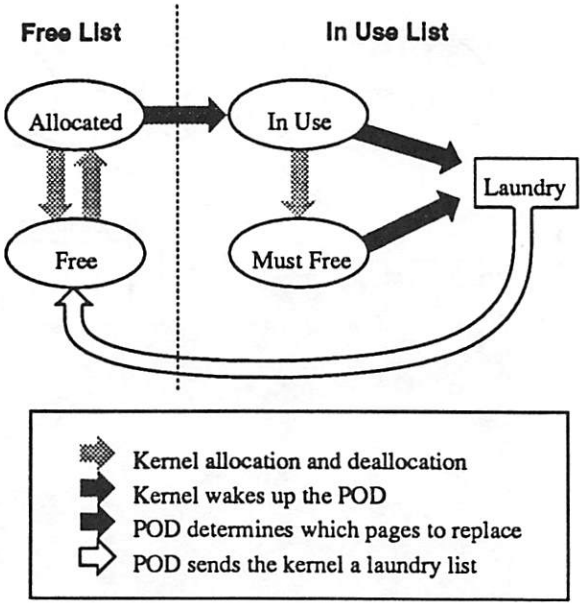


Figure 3: State diagram for page frame records

plicitly. First, since page records must be mapped into the kernel and into possibly several tasks, virtual address aliasing must be allowed by the machine architecture. While virtual address aliasing poses problems in some of today's caching architectures, we expect that future cache designs will address this problem [2, 13]. Second, because the POD only has access to the software-maintained machine-independent page records, and not the hardware-supported machine-dependent page tables, the POD will not have rapid access to the latest values of the modified and referenced bits. A kernel thread periodically updates the page-records. The absence of entirely up-to-date information is not expected to seriously impact the ability of PODs to manage memory effectively. Pages are flushed by the kernel, which does have access to the pmap interface, so recently modified pages will not be mistaken for clean pages. Third, the PODs share access to sensitive system data. They must therefore be cooperative and correctly written programs. Any ill-behaved POD could easily disrupt the system. This assumption of cooperation and correctness, however, must be made for the servers providing the infrastructure of any micro-kernel based system.

3.3 Sharing information between the kernel and PODs

In our architecture the resident page table and POD records are all mapped into the address space of every POD. Since PODs can both read and write sensitive data structures, it follows that PODs must be components of trusted systems, such as file and database management systems, rather than simple user applications. The delegation of responsibility to trusted processes is consistent with the microkernel

philosophy. It is not possible, however, to protect the kernel from the PODs or the PODS from one another.

The PODs operate on separate page frame lists and, when correctly programmed, do not access the same page frame records. Each POD, however, shares access to page frame records with the kernel and care must be taken that accesses are properly synchronized. Page frames are therefore divided between the free and in-use lists. The kernel does not access the in-use list. The POD does not access the free list. Transfers from and to the free list are accomplished by the creation of a temporary list whose head is passed to the POD or to the kernel via RPC. The POD accesses only the records of those frames on the in-use list. The kernel alone changes the state of a page frame on either list.

To implement a page replacement algorithm, a POD examines page frames on the in-use list. A thread in the kernel periodically updates the modified and referenced bits for these page records using information received through the pmap interface. For large page frame pools this information can be somewhat stale without significantly effecting the effectiveness of the approximate LRU algorithm. The kernel may also deallocate page frames on the in-use list (changing their state to must-free). The kernel does not alter the pointers of the in-use list.

We have made a few small additions to the Mach kernel interface to support user-level physical memory management. A POD registers with the kernel through the *phys_mem_request()* call. This call specifies the POD's minimum number of pages, the free-list low-water mark, and the periodic wakeup interval. The POD registers objects to be backed by its page frame pool by calling *phys_mem_register_object()*. The kernel wakes up PODs through the *phys_mem_pod_wakeup()* call. A POD can request that the kernel flush a list of pages through the *phys_mem_flush_queue()* call. The kernel removes each page from the object to which it is attached. If the page has been modified, the kernel invokes the pager through the current *memory_object_data_write()* interface.

3.4 Balancing demands for physical pages

The assignment of control over particular page frames to a POD is not permanent. Page frames can be reassigned to meet changing patterns of use. New PODs can also be created dynamically. The assignment of page frames to PODs is the responsibility of a *balance manager*. Each POD reflects its "satisfaction" with its current memory allotment through a parameter provided to the balance manager through the POD record. Each POD has a declared minimum page requirement (the POD cannot be started if this requirement cannot be met). Subject to these constraints, the balance manager can shift free page frames from the free-list of one POD to that of another. The computation of satisfaction parameters is specific to managers and their proper computation is a subject for future research. Likewise, the balancing

of their requests will have to be examined in future work. Our present mechanism simply provides the framework. In our initial design, the balance manager attempts to maximize a weighted sum of these satisfaction parameters.

4 Implementation and Performance

In our initial implementation we have created user-level PODs implementing simple page replacement strategies. While our primary purpose is to create physical memory managers for file and database management systems rather than for virtual memory systems, we can run our system using a user-level POD to manage backing store for virtual memory. In this case we change the default POD from a kernel thread to one running at the user level.

Our implementation required some modifications of existing kernel data structures and their use, as well as the addition of some new structures. The principal modifications include

- The kernel and PODs need to share two sets of data structures, the resident page table and POD records (as shown in Figure 1). To allow this sharing, an object containing both the resident page table and all POD records is created during physical memory initialization. This object is mapped into a POD's address space as part of its registration through *phys_mem_request()*. For simplicity we decided to keep the object's size static. This means that the number of POD records and the size of the resident page table remains fixed.

- In the original Mach kernel, a page frame can be found in one of four places: the free list, the active list, the inactive list, or a zone. In our kernel, a page frame belongs to a POD's free list, a POD's in-use list, or to a zone. The change in page frame states (and possible page moves) made it necessary to change the page frame data structure and a number of kernel routines.

- When waiting for a page frame to be initialized by a pager during a page fault, a *fictitious* page frame record is used as a place-holder pointed to by a memory cache object. A fictitious page frame record does not point to any physical page frame. In the original kernel, the page frame pointers of a real and a fictitious page frame records are swapped once a new page has been initialized, essentially exchanging the roles of the two records. For simplicity we decided to map only real page frame records into the PODs' address spaces. Fictitious and real page frame records therefore cannot change roles. Instead the record pointed to by the memory cache object is changed.

The principal additions to the kernel were the POD data structure and maps allowing the POD to be found.

- The POD records mentioned above contain the following information: POD's task, POD's port, free list pointer, free count, free list lock, transfer list pointer, in-use list pointer, minimum free list size, free list low water mark, restart interval, and time to next restart. In our current implementation, the 'restart interval' and 'time to next restart' are not used.

- When a pager registers an object with the kernel, the kernel creates an object–POD mapping (which is hashed according to the object). When the object is actually mapped into an address space and created, the kernel fills the object's POD field from this mapping. Then, when the kernel needs to service a page fault for the object, the kernel is able to allocate a page from the appropriate POD's pool.

Our initial implementation runs on a Sun 3/60 with 12 Mbytes of memory. For our system, the cost of faulting on a zero-filled page supplied by an external pager is about 3.2 ms. Consulting with the POD to find a free page at the time of the fault adds an over head of approximately 14%. This is on the order of the cost of consulting the external pager, and accords well with McNamee and Armstrong's 10% overhead for consulting the external pager to obtain a free page [6]. In our case, however, this overhead is a worst-case figure. Since the POD attempts to maintain a free-list available to the kernel, it need not be consulted synchronously on every page fault.

If a POD maintains a free list by freeing bursts of pages when invoked, most page faults can be handled without invoking the POD synchronously. Consequently, the cost of consulting a user-level POD rather than the current kernel page-out daemon has little effect on the time taken to run a significant application. A series of compiles while remaking the kernel, for example, consulted the POD between 35 and 39 times over five runs, each taking nearly three minutes of wall clock time and approximately 102 seconds of combined system and user time. Thus the POD is consulted only once every four to five seconds of real time and less than once every two to three seconds of compute time. Not surprisingly, the differences in times between several runs using the kernel page-out daemon and several runs with a user-level POD were negligible. Frequent accesses to the POD can be artificially induced by creating processes that access large amounts of memory randomly. In this case the critical factor is disk I/O activity, rather than context switches between kernel and POD.

5 Conclusions

Systems in the future will have to handle a more diverse workload than they do today. This workload will include managing very large data objects such as images, audio streams, and video streams, as well as an increasing number of sophisticated

object managers and their underlying data management systems. All of these systems will rely on sophisticated use of caching for performance. It is therefore important for operating systems to provide to incorporate mechanisms allowing diverse memory management policies to be implemented, and to provide mechanisms for mediating contention for memory among these subsystems.

Our design exposes important data structures to selected user-level processes. This allows subsystems to take a direct role in managing physical memory assigned to them. Our design provides an interface through which physical memory managers can monitor the use of the page frames assigned to them and determine which pages to flush. In other work, we are evaluating the demands placed on file system buffers when handling photographic image files [10] to test the usefulness of this interface. Our design allows memory to be shifted among pools at the command of a balance manager within the kernel. We are investigating appropriate balancing policies.

References

[1] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of VLDB '85*, pages 127–141, Stockholm, Sweden, 1985.

[2] James R. Goodman. Coherency for multiprocessor virtual address caches. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, October 1987.

[3] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Recovery management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82–108, February 1988.

[4] H. Levy and P. Lipman. Virtual memory management in the VAX/VMS operating system. *IEEE Computer*, 22(3):35–41, March 1982.

[5] Keith Loepere. *MACH 3 Kernel Principles.* Open Software Foundation and Carnegie Mellon University, 1991.

[6] Dylan McNamee and Katherine Armstrong. Extending the Mach external pager interface to accommodate user-level page replacement policies. In *Proceedings of the USENIX Association Mach Workshop*, pages 17–29, Burlington, Vermont (USA), October 1990. USENIX Association.

[7] Sun Microsystems. *SunOS 4.1 Programmer's Manual.* Sun Microsystems, Inc, Mountainview, California, 1990.

[8] Michael N. Nelson. Virtual memory vs. the file system. Research Report 90/4, Digital Western Research Laboratory, March 1990.

[9] James Van Sciver and Richard F. Rashid. Zone garbage collection. In *Proceedings of the USENIX Association Mach Workshop*, pages 1–15, Burlington, Vermont (USA), October 1990. USENIX Association.

[10] Stuart Sechrest, Khaled Charif, and Wu-Chi Feng. File block costs of zooming and panning in JPEG compressed images. Technical Report CSE-TR-98-91, University of Michigan, 1991.

[11] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[12] Michael Stonebraker. Managing persistent objects in a multi-level store. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 2–11, Denver, Colorado (USA), June 1991. ACM, New York (USA).

[13] Wen-Hann Wang, Jean-Loup Baer, and Henry M. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 140–148, Jerusalem, Israel, June 1989.

[14] James Wyllie. Personal communication, October 1991.

Page Replacement and Reference Bit Emulation in Mach

Richard P. Draves

rpd@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue Pittsburgh, PA 15213

Abstract

A page-replacement algorithm guides the operating system when it needs to create more free pages by reclaiming pages that are filled with data. The Mach kernel approximates the familiar Least Recently Used (LRU) algorithm with an algorithm known as FIFO with Second Chance. This strategy uses page-referenced information to avoid reclaiming recently-referenced pages. On hardware lacking such support, the kernel must detect references in software.

This paper describes the Mach kernel's page-replacement algorithm and considers three software techniques for detecting references. It shows that Mach 3.0's *reference fault* technique outperforms Mach 2.5's *reactivation fault* technique, and also outperforms reference detection via a software TLB miss handler. Based on the performance of the Mach 3.0 implementation, we conclude that hardware page-referenced information is not a prerequisite for a satisfactory page-replacement algorithm.

1 Introduction

The Mach kernel uses a simple global page-replacement algorithm, called FIFO with Second Chance, that requires page-referenced information to approximate LRU behavior. The Mach 2.5 implementation performed unsatisfactorily on architectures without hardware page-referenced support. The Mach 3.0 kernel uses machine-independent reference bits and *reference faults* to emulate page-referenced support. This implementation provides an adequate LRU approximation with low overhead, leading to the conclusion that hardware page-referenced information is not a requirement for a satisfactory page-replacement algorithm.

Page replacement is a very old problem [Belady 66, Denning 68] that has received little attention in recent years. This is for good reason. The choice of page-replacement algorithm largely doesn't matter, because memory is cheap and most machines have enough memory that paging is not a concern. If a machine clearly doesn't have enough memory to support its work-load, then no page-replacement algorithm can rescue performance. However, a page-replacement algorithm should still meet several criteria:

This research was supported in part by a fellowship from the Fannie and John Hertz Foundation and in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDS972-90-C-0035.

- *Low Overhead* — Because in many environments very little page-replacement occurs, the algorithm should not impose a noticeable overhead when it is not needed.

- *No Tuning* — An algorithm that requires a human other than the original designer to set parameters will perform poorly in the real world. Machine-dependent and user-dependent tuning is undesirable.

- *LRU Approximation* — When it is needed, the page-replacement algorithm should avoid pathological behavior. In a slightly memory-poor environment, an LRU approximation that keeps the system's working set resident will make a difference.

For example, as described later in Section 2.3, tuning Mach's page-replacement algorithm to produce a better LRU approximation improved the performance of a simple compilation test by a factor of four.

This paper describes the Mach kernel's page-replacement algorithm. In Section 2 we give an overview of the algorithm's operation and point out some problems with the emulation of hardware page-referenced information. We examine TLB-based emulation in Section 3 and then present Mach 3.0's machine-independent emulation in Section 4. In Section 5, we discuss the influence of Mach's external memory management interface on the page-replacement algorithm. We consider related work in Section 6. Finally, in Section 7 we summarize the paper.

2 Page Replacement in Mach

The Mach kernel uses a FIFO with Second Chance page-replacement algorithm. This algorithm approximates the performance of LRU page-replacement with the efficiency of FIFO page-replacement. The machine-independent pageout daemon, which implements the algorithm, uses Mach's pmap interface to manipulate machine-dependent page-referenced information. On architectures without such information, the pmap module and the pageout daemon must interact to emulate page-referenced support.

Mach's page-replacement algorithm has two major advantages:

- *Simplicity* — The algorithm attempts to reclaim a reasonable page, not the optimal page. A small amount of code and runtime effort produces good performance.

- *Scalability* — The performance of the algorithm is relatively insensitive to the actual amount of physical memory, because the pageout daemon never scans physical memory directly. Instead, it operates on queues of physical pages.

Unfortunately, the Mach 2.5 implementation of the algorithm proved to be difficult to tune properly on architectures without page-referenced support, and consequently it performed unsatisfactorily on those architectures.

2.1 FIFO with Second Chance

The Mach implementation of FIFO with Second Chance uses three queues of physical pages, known as the free queue, the active queue, and the inactive queue. The pageout daemon, a system thread running inside the kernel's address space, moves pages from the inactive queue to the free queue and from the active queue to the inactive queue.

To manage physical memory, the kernel manipulates small page structures. Each page structure represents one page of physical memory. The structure records various attributes of the physical page, including its address, and contains link fields for the paging queues.

Pages on the free queue contain no useful data. All pages on the free queue are equivalent. The kernel allocates pages from the free queue to satisfy page faults or to perform internal data structure allocation. Zero-filling, if necessary, happens after a page is taken from the free queue. If there are insufficient free pages, an allocating thread waits for the pageout daemon to create more free pages.

Pages on the active queue are mapped by some process and are assumed to belong to the system's working set. The active queue is FIFO; new pages start at the back of the queue and the pageout daemon takes pages from the front of the active queue.

The inactive queue, which is also FIFO, acts as a buffer between the free queue and the active queue. Inactive pages are assumed to be less valuable than active pages. Pages which are referenced while on the inactive queue return to the active queue instead of moving to the free queue. The inactive queue implements the page-replacement algorithm's LRU approximation by preventing frequently-referenced pages from being reclaimed.

The pageout daemon's primary responsibility is creating free pages. It wakes up when the free queue falls below `vm_page_free_min` pages, and runs until the free queue contains at least `vm_page_free_target` pages. The kernel uses these thresholds to reduce the overhead of repeatedly invoking the pageout daemon. Once awake, the pageout daemon scans the inactive queue. Clean pages are immediately moved to the free queue, and dirty pages are first cleaned by writing them to backing store. After the free queue reaches `vm_page_free_target` (or if the inactive queue is empty), the pageout daemon refills the inactive queue from the active queue, until it reaches `vm_page_inactive_target`.

Figure 1 shows the movement of pages among the active, inactive, and free queues.

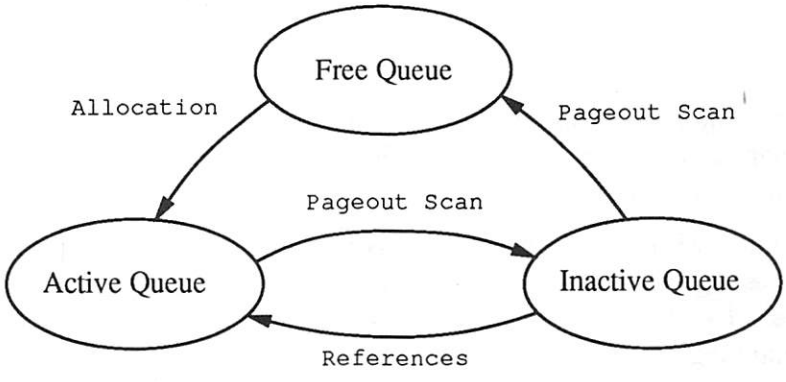


Figure 1: The Paging Queues

2.2 Reference Bits

The FIFO with Second Chance algorithm relies on being able to detect references to pages on the inactive queue. The Mach kernel makes use of page-referenced information, if it is provided by the hardware, through two pmap functions. If the hardware does not directly supply page-referenced information then the kernel must somehow emulate it. As done in Mach 2.5, this leads to two very different modes of operation for the page-replacement algorithm.

The Mach kernel accesses MMU hardware through a machine-independent interface, the pmap interface [Rashid et al. 87, Tevanian 87]. A pmap is the machine-dependent representation of an address space. The pmap interface contains, for example, functions to create and destroy pmaps and enter virtual-to-physical mappings into a pmap. The kernel's pmap module implements the pmap interface.

Some hardware architectures provide page-referenced information. On architectures with page tables, this commonly takes the form of a reference bit in each page-table entry. The operating system can clear the bit in a page-table entry and virtual memory references through a page-table entry set the bit. However, other forms of page-referenced information are possible. For example, the inverted page table found in the IBM RT [IBM 88] has a single reference bit per physical page.

The pmap module must provide two functions on a physical page address, to clear and query reference information, as shown in Figure 2. Because the interface operates on physical pages instead of virtual pages, the implementation on architectures with reference bits in page-table entries must examine or modify the reference bit in every page-table entry mapping the specified physical page. On architectures without hardware page-referenced information, `pmap_clear_reference` should remove all mappings for the physical page and `pmap_is_referenced` should always return FALSE.

`void pmap_clear_reference(physical address)`
: Clear machine-dependent page-referenced information associated with the specified physical page.

`boolean_t pmap_is_referenced(physical address)`
: Query machine-dependent page-referenced information associated with the specified physical page, returning TRUE if the physical page has been accessed since the page-referenced information was last cleared.

Figure 2: Page-Referenced Functions

Using these pmap functions, the Mach 2.5 pageout daemon functions as shown in Figure 3. The page-replacement algorithm behaves quite differently on machines with and without hardware page-referenced support:

- On machines without such support, pages on the inactive queue are not mapped into any address space. Faults on pages in the inactive queue, known as *reactivation faults*, are satisfied without performing I/O, by moving the page back to the active queue and remapping it.
- On machines with such support, pages on the inactive queue are left mapped into address spaces. Instead of using reactivation faults to move pages back to the active queue, the pageout daemon notices before freeing a page that it has been referenced, and activates the page instead. This type of reactivation may be viewed as the lazy evaluation of reactivation faults.

2.3 A Problem

Mach's first platform was the VAX-11/784, a four-processor version of the VAX-11/780. The VAX architecture [DEC 79] does not provide hardware reference bits. As originally conceived [Tevanian 87, pages 43–45], the inactive queue contained pages not mapped into any

```
vm_pageout_scan() {
        while (vm_page_free_count < vm_page_free_target) {
                page = dequeue(vm_page_inactive_queue);
                if (pmap_is_referenced(page->phys_addr)) {
                        enqueue(vm_page_active_queue, page);
                } else {
                        clean the page if it is dirty;
                        enqueue(vm_page_free_queue, page);
                }
        }

        while (vm_page_inactive_count < vm_page_inactive_target) {
                page = dequeue(vm_page_active_queue);
                pmap_clear_reference(page->phys_addr);
                enqueue(vm_page_inactive_queue);
        }
}
```

Figure 3: The Mach 2.5 Pageout Daemon

address space, and at boot-time the implementation configured `vm_page_inactive_target` to a small value, never larger than 80 pages. Once configured, the implementation never changed `vm_page_inactive_target`.

After some experience with the page-replacement algorithm, it became clear that the original configuration of the `vm_page_inactive_target` was not appropriate for machines with hardware page-referenced support. In fact, it was desirable to make the inactive queue larger than the active queue. A large inactive queue improves the quality of the LRU approximation by giving pages more time in which to be referenced. When the paging rate is high and the pageout daemon runs frequently, a small inactive queue is ineffective because relatively few pages are reactivated. With a small inactive queue and a high paging rate, the page-replacement algorithm degenerates to almost pure FIFO replacement. However, a reasonable LRU approximation is most important when the paging rate is high, because it is only by finding the system's working set and keeping it resident that the paging rate will come down.

To examine this effect, we used a small compilation test suite on an DECstation 3100 (DS3100) configured to use 8 megabytes of memory. The DS3100 has a 16.67Mhz MIPS-architecture R2000 CPU. The MIPS architecture does not provide page-referenced information. The compilation test compiles nine small C programs.

We ran the compilation test nine consecutive times while varying the inactive target. Figure 4 summarizes the results. With a small inactive target of 20 pages out of 1100 available pages, the Mach 2.5 algorithm performed poorly, leading to excessive paging. With an inactive target of 700 pages, the test completed approximately four times faster. Even after the inactive target was lowered to the original small value, the test completed more quickly because the smaller inactive queue became more effective once the paging rate had dropped. For comparison, the test took 19.2s to complete when run single-user on a DS3100 with 16 megabytes of memory,

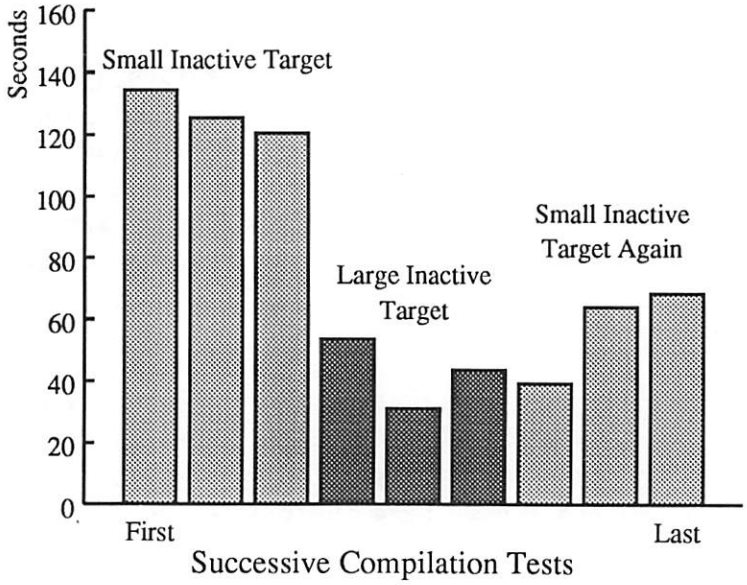


Figure 4: A Small Inactive Target Leads To Thrashing

Unfortunately, a large inactive queue is not a panacea. On machines without hardware reference bits, pages on the inactive queue are unmapped. A large inactive queue on these machines reduces the amount of physical memory directly available to applications, leading to *reactivation fault thrashing.* This can occur when an application's working set resides in physical memory but exceeds the capacity of the active queue. On the DS3100, for example, a reactivation fault takes 115μs. Hence, inappropriate tuning of the page-replacement algorithm can greatly decrease an application's performance.

Furthermore, an extremely large inactive queue increases the overhead of the pageout daemon. For each unreferenced page that it finds on the inactive queue, it must pass up some number of referenced pages. If the inactive queue contains most of physical memory and almost all pages on it are referenced, then the page-replacement algorithm's performance degenerates. To find an unreferenced page, the pageout daemon will search the large inactive queue, moving referenced pages from the inactive queue to the active queue and then back to the inactive queue to replenish it.

The solution to this dilemma in Mach 2.5 was to make it possible for machine-dependent code to override the boot-time configuration of `vm_page_inactive_target` and the other parameters of the page-replacement algorithm. The default size of the inactive queue was increased to two-thirds of available memory, and architectures without hardware reference support overrode this to use approximately 2% of available memory.

3 TLB-Based Reference Bit Emulation

Some architectures, like the MIPS architecture [Kane 88] used in the DECstation 3100, provide a TLB which requires software refill. Because the operating system supplies the TLB refill handler, these architectures present considerable flexibility for the operating system to tailor TLB refill to its needs. For example, TLB refill may be used to provide page-referenced information. We implemented this successfully in Mach, but at the cost of increased TLB miss latency.

We experimented with having the MIPS TLB miss handler emulate reference bits. When emulating reference bits, the MIPS pmap module maintains an array of reference bits, one per physical page. For efficiency, each reference bit is stored in a separate byte and a non-zero byte indicates the corresponding physical page has *not* been referenced. The pmap operations `pmap_clear_reference` and `pmap_is_referenced` index into the array and manipulate the appropriate byte. The TLB miss handler clears the byte for a physical page when loading a mapping for that page into the TLB. This modification to the miss handler increases the latency of servicing a miss from 9 cycles to 16 cycles.

With this technique, the MIPS architecture appears to support page-referenced information just like architectures with true support in hardware. The pmap interface hides the difference from the machine-independent page-replacement algorithm. Consequently, with reference bit emulation enabled the MIPS architecture can configure a large inactive queue and enjoy the benefits of an improved LRU approximation.

However, the latency of TLB refill is critical to overall system performance. For example, we counted the number of TLB misses generated by the compilation test of Section 2.3, run single-user on a DS3100 with 16 megabytes of memory and an instrumented kernel. In this environment, the compilation test has enough available memory to run to completion without page replacement. On average, the compilation test invoked the TLB miss handler 560,000 times. Given the 16.67Mhz clock rate of the DS3100 and the increase in latency of the miss handler, we can calculate

$$560000 * (16 - 9)/16.67\text{Mhz} = 0.23\text{s}$$

the performance degradation expected for reference bit emulation. To confirm this effect, we also timed the compilation test. With reference bit emulation, the test completed in 19.4s. Without reference bit emulation, the test completed in 19.2s.

Overall, reference bit emulation via TLB misses was an improvement because it let us increase the size of the inactive queue and reduce paging. However, this solution did not apply to other architectures lacking hardware page-referenced information, like the VAX, and it cost about 1% in overall performance when no paging was occurring.

4 Machine-Independent Reference Bit Emulation

The Mach 3.0 implementation of the FIFO with Second Chance page-replacement algorithm uses a machine-independent reference bit and reference faults to emulate hardware reference bits. With this modification, the algorithm operates similarly whether or not the hardware architecture supplies page-referenced information. Reactivation fault thrashing is no longer a concern.

The reference fault mechanism extends the machine-independent page structure with a reference bit. As before, when the pageout daemon moves a page from the active queue to the inactive queue, it uses `pmap_clear_reference` to clear machine-dependent page-referenced information, or remove all mappings for the page if the pmap module does not support reference information. At this time, the page's machine-independent reference bit is also cleared. When a fault on an inactive page occurs, the fault handler does *not* activate the page. Instead, it sets the machine-independent reference bit, creates a mapping for the page, and leaves the page on the inactive queue. These faults are known as *reference faults*. Before the pageout daemon moves a page from the inactive queue to the free queue, it uses the machine-independent reference bit and `pmap_is_referenced` to check

both machine-independent and machine-dependent sources of page-referenced information. Figure 5 outlines the modified algorithm.

```
vm_pageout_scan() {
        while (vm_page_free_count < vm_page_free_target) {
                page = dequeue(vm_page_inactive_queue);
                if (page->referenced ||
                    pmap_is_referenced(page->phys_addr)) {
                        enqueue(vm_page_active_queue, page);
                } else {
                        clean the page if it is dirty;
                        enqueue(vm_page_free_queue, page);
                }
        }

        while (vm_page_inactive_count < vm_page_inactive_target) {
                page = dequeue(vm_page_active_queue);
                pmap_clear_reference(page->phys_addr);
                page->referenced = FALSE;
                enqueue(vm_page_inactive_queue);
        }
}
```

Figure 5: The Mach 3.0 Pageout Daemon

This modification unifies the operation of the page-replacement algorithm on architectures with and without page-referenced support. In both cases, references to pages on the inactive queue leave the page on the queue until the pageout daemon notices that the page has been referenced. In both cases, pages on the inactive queue may be mapped into address spaces. Therefore, the reference fault mechanism eliminates the need to tune the algorithm for different architectures. Machine-independent code can configure the size of the inactive queue.

Because pages on the inactive queue can be mapped into address spaces, architectures without page-referenced support no longer suffer from reactivation fault thrashing when the inactive queue is large. If the system's working set exceeds the size of the active queue but fits in memory, then the system reaches a quiescent state in which most of the inactive pages are marked as referenced and are mapped.

Allowing inactive pages to be mapped in effect allows the size of the inactive queue to change dynamically. When the system is paging heavily, few pages on the inactive queue are referenced. When the paging rate drops because the system's working set is resident, then most pages in the inactive queue become referenced and the effective size of the inactive queue decreases. The advantage of this method over explicitly adjusting `vm_page_inactive_target` for difference performance regimes is that the changes in the composition of the inactive queue happen automatically.

5 Page Replacement and External Memory Management

The pageout daemon faces two related problems, avoiding deadlocks and flow control, in addition to its primary problem of choosing pages to remove from main memory. The original page-replacement implementation proceeded the implementation of the external memory manager (XMM) interface [Rashid et al. 87, Young 89], and this greatly simplified the pageout daemon. The Mach 3.0 pageout daemon uses several strategies to cope with slow or uncooperative external memory managers.

When the original pageout daemon cleaned dirty pages, it copied them into the I/O system's buffer cache. The size of the cache automatically limited the rate at which the daemon could clean pages, because once it filled with data waiting to be written to disk, the daemon would start waiting for disk writes to complete and free buffers.

In contrast, the Mach 3.0 pageout daemon sends dirty pages to memory managers in `memory_object_data_write` messages. Because the XMM interface is asynchronous, the pageout daemon does not know when a memory manager has completed processing a `memory_object_data_write`. If the pageout daemon encounters a long sequence of dirty pages on the inactive queue and makes no attempt at flow control, it can generate arbitrarily long queues of `memory_object_data_write` messages waiting in the IPC system [Draves 90]. The normal queue-limiting flow control mechanisms of the IPC system are not appropriate (and are not applied) because the pageout daemon can't wait for a potentially buggy or malicious memory manager to take some action.

The Mach kernel does rely on a distinguished memory manager, the default memory manager [Golub & Draves 91], to operate correctly. The default memory manager services memory objects created when a process allocates temporary memory, not managed by an explicitly specified memory manager. The default memory manager also acts as the "pager of last resort," which cleans the dirty pages that unprivileged memory managers fail to process.

The pageout daemon and the default memory manager on occasion must allocate memory, consuming free pages, while they operate to create more free pages. For example, the pageout daemon allocates message buffers for `memory_object_data_write` messages. To prevent a deadlock when there are no free pages, the kernel maintains a reserve of free pages and only the pageout daemon and default memory manager can allocate free pages from the reserve. If the pageout daemon fails at flow control and generates `memory_object_data_write` messages more quickly than they are processed, then it can consume the free page reserve and deadlock.

The Mach 3.0 kernel uses three thresholds to prevent the free page queue from being exhausted:

`vm_page_free_reserved`
: Below this level, only privileged threads can allocate free pages.

`vm_pageout_reserved_internal`
: Below this level, the pageout daemon no longer expects memory managers other than the default memory manager to function. The pageout daemon cleans pages associated with other memory managers by double-paging them, or sending them first to their real memory manager and then immediately resending them to the default memory manager.

`vm_pageout_reserved_really`
: Below this level, the pageout daemon stops operating and waits for the default memory manager to catch up.

Artificially constructed paging test programs can drive the size of free queue below the `vm_pageout_reserved_really` threshold, but the `vm_pageout_reserved_internal` threshold is rarely exceeded in practice.

The Mach 3.0 pageout daemon uses two strategies for flow control, to control separately the rate at which dirty pages are sent to the default memory manager and to other memory managers. Because of the constraints placed on the default memory manager, that it must be correct and it must free pages after cleaning them, the pageout daemon can use a reliable strategy with it. The strategy employed with other memory managers is only heuristic, but the `vm_pageout_reserved_internal` threshold described earlier prevents this from compromising system correctness.

To avoid flooding the default memory manager, the kernel keeps a "laundry count" of how many pages the default memory manager has not yet cleaned. Pages sent to the default memory manager are marked with a "laundry bit," and when pages so marked are returned to the free queue, the laundry count is decremented. The pageout daemon pauses to let the default memory manager catch up when the laundry count exceeds a threshold, `vm_pageout_burst_max`.

To avoid flooding other memory managers, the pageout daemon also pauses after sending out `vm_pageout_burst_max` dirty pages. Because the kernel does not have an equivalent of the laundry count for these memory managers, the pageout daemon must assume that its pause gave the memory managers time to process the `memory_object_data_write` requests.

The two flow control strategies work best when the pageout daemon's pauses are properly tuned. Instead of requiring machine-dependent tuning, the pageout daemon tunes the pause interval dynamically. If the pageout daemon wakes from a pause and the laundry count is still high, the pageout daemon increases the pause interval. If the laundry count is low after each of several consecutive pauses, the pageout daemon decreases the pause interval.

6 Related Work

Other operating systems for the VAX architecture, notably VMS and BSD Unix, perform page-replacement without the benefit of hardware page-referenced information. Some researchers have taken a different approach to page replacement in Mach and have experimented with application-level control over page-replacement decisions.

The VAX/VMS operating system[Goldenberg & Kenah 91, Turner & Levy 81] uses per-process working sets with global free and modified lists to control page replacement. Normal FIFO page replacement occurs within a working set, each of which behaves like Mach's active queue. The global free and modified lists cache pages not mapped into processes. Page faults which are satisfied from the free and modified lists without I/O are known as *soft page faults*. Like Mach's reactivation faults, they ameliorate the FIFO replacement within working sets to produce an LRU approximation. They also suffer from the same performance tuning dilemma. For example, [Lazowska 79] found that the default size for the free and modified lists was too small for memory-poor environments.

The BSD Unix operating system uses a clock algorithm [Babaoglu & Joy 81]. The clock hand cycles around main memory, clearing software reference bits and unmapping pages. Faults on unmapped pages, like Mach's reference faults, set the software reference bit. The clock algorithm reclaims unreferenced pages found by the hand. Unlike Mach's algorithm, the BSD algorithm suffers from scalability problems, because the clock hand must sweep across the machine's entire physical memory. It is also difficult to tune properly the speed

of the clock hand.

Some researchers [McNamee & Armstrong 90] have taken the approach of providing control over page-replacement to the application. This is a promising approach for applications like Lisp, ML, and databases, but it leaves open the question of a default mechanism. Mach's FIFO with Second Chance algorithm can easily accommodate simple application-level hints about the relative worth of pages. Pages deemed less valuable can be forcibly moved from the active queue to the inactive queue to make them more likely candidates for replacement.

7 Conclusions

The Mach kernel uses a FIFO with Second Chance page-replacement algorithm. We have experimented with several approaches to providing the page-referenced information that the algorithm needs to approximate LRU behavior. An approach based on a machine-independent reference bit and reference faults outperforms Mach 2.5's reactivation fault technique and reference bits based on software TLB refill. The FIFO with Second Chance algorithm can accommodate the requirements of external memory management.

Acknowledgements

Avadis Tevanian and Michael Young first implemented page-replacement in the Mach kernel. Alessandro Forin implemented the TLB-based reference bit emulation described in Section 3. Michael Young implemented preliminary versions of the mechanisms described in Section 5.

References

[Babaoglu & Joy 81] Babaoglu, O. and Joy, W. Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Referenced Bits. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 78–86, December 1981.

[Belady 66] Belady, L. A. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[DEC 79] Digital Equipment Corporation. *VAX-11 Architecture Handbook*, 1979.

[Denning 68] Denning, P. J. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[Draves 90] Draves, R. P. A Revised IPC Interface. In *Proceedings of the USENIX Mach Workshop*, pages 101–121, October 1990.

[Goldenberg & Kenah 91] Goldenberg, R. E. and Kenah, L. J. *VAX/VMS Internals and Data Structures: Version 5.2.* Digital Press, 1991.

[Golub & Draves 91] Golub, D. B. and Draves, R. P. Moving the Default Memory Manager out of the Mach Kernel. In *Proceedings of the Second USENIX Mach Symposium*, November 1991. This issue.

[IBM 88] International Business Machines, Austin, Texas. *IBM RT PC Hardware Technical Reference Volume 1*, third edition, 1988.

[Kane 88] Kane, G. *MIPS RISC Architecture.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Lazowska 79] Lazowska, E. D. The Benchmarking, Tuning and Analytic Modeling of VAX/VMS. In *Papers Presented at the 1979 Conference on Simulation, Measurement and Modeling of Computer Systems*, pages 57–64, August 1979.

[McNamee & Armstrong 90] McNamee, D. and Armstrong, K. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. In *Proceedings of the USENIX Mach Workshop*, pages 17–29, October 1990.

[Rashid et al. 87] Rashid, R., Tevanian, Jr., A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, October 1987.

[Tevanian 87] Tevanian, Jr., A. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach.* PhD dissertation, Carnegie Mellon University, December 1987.

[Turner & Levy 81] Turner, R. and Levy, H. Segmented FIFO Page Replacement. In *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, pages 48–51, September 1981.

[Young 89] Young, M. W. *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.* PhD dissertation, Carnegie Mellon University, November 1989.

Evaluation of Real-Time Synchronization in Real-Time Mach

Hideyuki Tokuda and Tatsuo Nakajima
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
hxt@cs.cmu.edu

Abstract

Real-Time Mach provides real-time thread and real-time synchronization facilities. A real-time thread can be created for a periodic or aperiodic activity with a timing constraint. Threads can be synchronized among them using a real-time version of the monitor based synchronization mechanism with a suitable locking protocol. In Real-Time Mach, we have implemented several locking policies, such as *kernelized monitor*, *basic priority priority inheritance protocol*, *priority ceiling protocol*, and *restartable critical section*, for real-time applications. It can also avoid a unbounded *priority inversion* problem.

In this paper, we describe the real-time synchronization facilities in Real-Time Mach and its implementation and performance evaluation. Our evaluation results demonstrated that a proper choice of locking policy can avoid unbounded priority inversions and improve the processor schedulability for real-time applications.

1 Introduction

A new challenge in advanced real-time systems is not only creating a fast and responsive kernel, but providing a predictable and analyzable real-time computing environment. An advanced real-time kernel should allow a system designer to analyze the runtime behavior at the design stage and predict whether the given real-time tasks having various types of system interactions can meet their timing requirements.

The objective of Real-Time Mach is to create a real-time version of Mach operating system which can provide such predictable real-time computing environment. However, in the Mach kernel, it is often difficult to analyze the runtime behaviour of the time critical activities due to lack of real-time scheduling and synchronization facilities. For instance, Mach provides synchronization facility among threads using a conventional monitor-based mechanism[5]. Threads can enter a critical section in FIFO order, however, this may cause a *a priority inversion problem* among threads. If many lower priority threads are already waiting on a mutex variable, a higher priority thread which may have a tighter deadline must wait for their completion. Thus, the higher priority thread may miss its deadline.

Furthermore, a real-time system designer need to determine the worst case blocking delay for a higher priority thread for the shared resource. It is often impossible to compute the bound if a thread in the protected region can be preemptable. Let us consider the following case. Suppose that the lowest priority thread T_L is in the critical region first, then the highest priority thread T_H becomes runnable and attempts to obtain the mutex variable. However, since T_L is in the critical region, T_H must wait for its completion. After T_L resumed, a medium priority thread T_{M1} becomes runnable. Then T_{M1} starts running without using the critical resource and wakes up another medium priority thread T_{M2} and so on. Under this type of interactions, the worst case blocking time of T_H cannot be determined by without knowing all behavior of related medium priority thread T_M's.

In order to bound the worst case blocking time, a simple solution called *priority inheritance* scheme was developed in our group [12, 10, 14]. An priority inheritance scheme is that once T_H blocks on the mutex variable, T_L inherits the priority of T_H. Then, T_{M1} cannot preempt the activity of T_L in the critical region. In this way, the worst case blocking time of T_H can be a function of the duration of the critical region, and not a function of the execution times of the medium priority tasks.

Real-Time Mach [15] provides a set of locking protocols for the mutex variable for sharing resources among real-time threads. We have implemented five locking protocols, namely *kernelized monitor*, *basic priority*, *basic priority inheritance*

[1] This research was supported in part by the U.S. Naval Ocean Systems Center under contract number N66001-87-C-0155, by the Office of Naval Research under contract number N00014-84-K-0734, by the Defense Advanced Research Projects Agency, ARPA Order No. 7330 under contract number MDA72-90-C-0035, by the Federal Systems Division of IBM Corporation under University Agreement YA-278067, and by the SONY Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of NOSC, ONR, DARPA, IBM, SONY, or the U.S. Government.

protocol, *priority ceiling protocol*, and *restartable critical section* to be used for various real-time applications.

In this paper, we describe the implementation and performance evaluation of real-time synchronization facilities in Real-Time Mach. In Section 2, we first introduce the real-time thread model and synchronization facilities in Real-Time Mach. Section 3 discusses the extended system interface for real-time synchronization and the implementation of synchronization mechanisms and policy modules. In Section 4, we also describe the performance evaluation of thread preemption and the locking protocols. Related work is discussed in Section 5 and we describe the conclusion and future work in Section 6.

2 Real-Time Thread and Synchronization Model

In this section, we briefly describe a real-time thread and synchronization model we adopted in Real-Time Mach and the notion of schedulable bound for synchronizing real-time tasks in a single processor environment.

2.1 Real-Time Thread Model

A thread can be defined for a real-time or non-real-time activity. Each thread is specified by at least a procedure name and a stack descriptor which specifies the size and address of the local stack region. For a real-time thread, additional *timing attributes* must be defined by a timing attribute descriptor. A real-time thread can be also defined as a *hard* real-time or *soft* real-time thread. By a hard real-time thread, we mean that the thread must complete its activities by its *hard* deadline time, otherwise it will cause undesirable damage or a fatal error to the system. The soft real-time thread, on the other hand, does not have such a hard deadline, and it still makes sense for the system to complete the thread even if it passed its critical (i.e. *soft* deadline) time.

A real-time thread can be also defined as a *periodic* or *aperiodic* thread based on the nature of its activity. A periodic thread P_i is defined by the worst case execution time C_i, period T_i, start time S_i, phase offset O_i, and task's semantic importance value V_i. In a periodic thread, a new instantiation of the thread will be scheduled at S_i and then repeat the activity in every T_i. The phase offset is used to adjust a ready time within each period. If a periodic thread is a soft real-time thread, it may need to express the abort time which tells the scheduler to abort the thread. An aperiodic thread AP_j is defined by the worst case execution time C_j, the worst case interarrival time A_j, deadline D_j, and task's semantic importance value V_i. In the case of soft real-time threads, A_j indicates the average case interarrival time and D_j represents the average response time. Abort time can be also defined for the soft real-time thread.

2.2 Real-Time Synchronization Model

In a real-time synchronization model, we have created various synchronization policies based on two basic factors: one is a queueing order among waiting threads and the other is preemptability of the running thread in the critical section.

Traditional synchronization primitives use FIFO ordering among waiting threads to enter a critical section, since FIFO ordering can avoid the starvation, In real-time computing environment, however, FIFO ordering often creates a priority inversion problem. A higher thread must wait for the completion of all low priority threads in the waiting queue. If all of real-time threads can meet their deadlines, then there will be no starvation among these threads. Thus, in real-time synchronization, the system should provide a deadline based (or priority based) ordering to avoid unbound priority inversion problems.

Preemptability of the running thread in the critical section also affects the synchronization policies and the schedulability of the task sets. In Real-Time Mach, the following three preemption levels of the running task in the critical section has been supported.

Non Preemptable: No preemption is allowed while a thread is executing in the critical section.

Preemptable: A higher priority thread can preempt the current running thread. If the higher priority thread need to enter the critical section, it will be blocked (i.e., it must wait for the completion of the critical section).

Restartable: A higher priority thread can preempt the current running thread. It then aborts the running thread and puts the thread back to the waiting queue. The higher priority thread executes the critical section without further delay. The preempted lower priority thread will restart later from the beginning of the critical section.

By selecting a queueing ordering and the above preemption choices, the following synchronization policies can be defined:

Kernelized Monitor protocol (KM): KM protocol takes the non preemptable mode.

Basic Priority protocol (BP): BP protocol takes the preemptable mode and the queueing ordering is based on the thread priority.

Basic Priority Inheritance protocol (BPI): BPI protocol is created as BP protocol plus the priority inheritance function. By this function, a lower priority thread executing the critical section inherits the priority of higher priority thread, when the lock is conflicted.

Priority Ceiling protocol (PCP): PCP protocol is an extension of BPI protocol. In PCP, the it ceiling priority of the lock is defined by the priority of the highest priority thread that may lock the lock variable. The execution of

the thread is blocked when the priority ceiling of the lock is not higher than all locks which are owned by other threads. The protocol prevents deadlock, and chained blocking. The underlying idea of PCP is to ensure that when a thread T preempts the critical section of another thread S and executes its own critical section CS, the priority at which CS will be executed is guaranteed to be higher than the inherited priorities of all the preempted critical sections.

Restartable Critical Section protocol (RCS): RCS policy is based on the restartable mode. In RCS, a higher priority thread is able to abort the current running thread in the critical section and puts it back to the waiting queue with recovering the state of shared variable. During the recovery, the higher priority thread's priority is inherited like in priority inheritance protocol. After this recovery action, the higher priority thread can enter the critical section without any waiting in the queue. A user program must be responsible to recover the state of shared variable.

Each synchronization protocol has a different characteristics for schedulability and a different cost to enter to and exit from critical sections. In the following section, we will show the difference between synchronization protocols. The result allows application programmer to select suitable synchronization protocols for their applications.

2.3 Real-Time Synchronization Analysis

Analyzing the schedulable bound for real-time tasks is a very difficult problem. We have developed a simple yet very applicable scheme for analyzing real-time periodic threads which are synchronizing in a single processor environment[1].

To compute a schedulable bound, we must avoid a potential unbounded priority inversion problem among real-time threads. Once we could bound the worst case blocking time, then we can compute a bound as an extension of the *rate monotonic* scheduling analysis [8].

There are basically two approaches to bound the worst case blocking time among real-time tasks. One is using a *kernelized monitor* protocol and the other is using a *priority inheritance* scheme as we described in the previous section.

In the kernelized monitor protocol, while a task is executing in a critical section, the system will not allow any preemption of that task. Suppose that if n tasks are scheduled in the earliest deadline first order, the worst case schedulable bound is defined as follow.

$$\sum_{j=1}^{n} \frac{C_j+CS}{T_j} \leq 1$$

where C_i, T_i, CS represents the total computation time of $Thread_i$, the period of $Thread_i$, and the worst case execution time of the critical section respectively. In general, if the duration of CS is too big, the system cannot satisfy the schedulability test and end up with reducing the number of tasks and running with a very low total processor utilization.

On the other hand, if we relax the kernelized monitor and allow a preemption during the critical section, we may face the unbounded priority inversion problem. Under a priority inheritance scheme, once the higher priority task blocks on the critical section, the low priority task will inherit the high priority from the higher priority task. One of extended priority inheritance protocols is called a *priority ceiling protocol*[10].

Using these inheritance protocols under a rate monotonic policy, we can also check schedulable bound for n periodic threads as follows.

$$\forall i, 1 \leq i \leq n, \quad \frac{B_i}{T_i} + \sum_{j=1}^{i} \frac{C_j}{T_j} \leq i(2^{\frac{1}{i}} - 1)$$

where C_i, T_i, B_i represents the total computation time, the period, and the worst case blocking time of $Thread_i$ respectively.

Restartable critical section protocol makes the blocking time to be smallest among the proposed protocols, if abort and recovery overhead is negligible. The schedulability formula for restartable critical section is similar to the formula for the priority ceiling protocol. The difference is that B_i should represent the abort and recovery time when using restartable critical section.

3 Implementation

The current version of Real-Time Mach is being developed using a network of SUN, SONY workstations, laptop and single board target machines. The pure kernel provided us a better execution environment where we could reduce unexpected delays in the kernel. The preemptability of the kernel was also improved significantly since UNIX primitives and some device drivers are no longer in the kernel. In this section, we will describe the extended system interface for real-time synchronization and the implementation of synchronization mechanisms and policy modules.

3.1 System Interface for Synchronization

The synchronization mechanism is based on mutual exclusion with a lock variable [2]. A thread can allocate, deallocate, and initialize a lock variable with a suitable *lock attribute*. The lock attribute specifies a synchronization policy which determines its queueing and priority ordering.

A simple pair of *rt_mutex_lock* and *rt_mutex_unlock* primitives is used to specify mutual exclusion. The *rt_mutex_trylock*

[1] For a multiprocessor case, a simple priority inheritance case may not work and further extensions are necessary. Please refer[10].

[2] We have also created event-based primitives: **rt_event_send**(event, event_attr), **rt_event_receive**(event, event_attr, timeout). However, we do not describe these primitives in this paper.

primitive is used for acquiring the lock conditionally. A modified version of the condition variable is also supported for specifying a conditional critical region. A pair of *rt_condition_signal* and *rt_condition_wait* primitives is used to synchronize over a condition variable.

The interface of the synchronization functions and lock attribute are summarized as follows.

```
kval_t = rt_mutex_allocate(lock, lock_attr)
kval_t = rt_mutex_deallocate(lock)
kval_t = rt_mutex_lock(lock, timeout, context)
kval_t = rt_mutex_unlock(lock)
kval_t = rt_mutex_trylock(lock)
kval_t = rt_condition_allocate(cond, cond_attr)
kval_t = rt_condition_deallocate(cond)
kval_t = rt_condition_wait(cond, lock, cond_attr, timeout)
kval_t = rt_condition_signal(cond, cond_attr)
```

```
typedef struct lock_attr {
    type_t          rt_type;        /* lock type */
    priority_t      rt_priority;    /* ceiling priority */
                    ...
} lock_attr_t;

typedef struct cond_attr {
    type_t          rt_type;        /* condition variable type */
    priority_t      rt_priority;    /* for priority inheritance */
                    ...
} cond_attr_t;
```

rt_type indicates a lock policy given by a user. *rt_priority* is used for the ceiling protocol and specifies the ceiling priority of the lock. If NULL value is set as a lock attribute, a default policy (i.e., BP, basic priority policy) is chosen. In *cond_attr*, *rt_type* indicates the type of condition variable, and *rt_priority* is used for priority inheritance.

3.2 Synchronization Module

The synchronization facility is implemented based on the policy/mechanism separation concept for improving the adaptability of the system. Each synchronization policy module is implemented as an object (or an abstract data type). The communication between the policy object and its mechanism was done through function calls, not message passing.

The synchronization module is divided into the common lock object and lock policy objects. Figure 1 shows the relationship between the common lock object and various lock policy objects. Only one common lock object is created, while the lock policy object is created for each lock object and controls the priority of the thread which is holding the lock object.

The common lock object offers the mechanism to manage the blocking and unblocking of threads for exclusive access It also

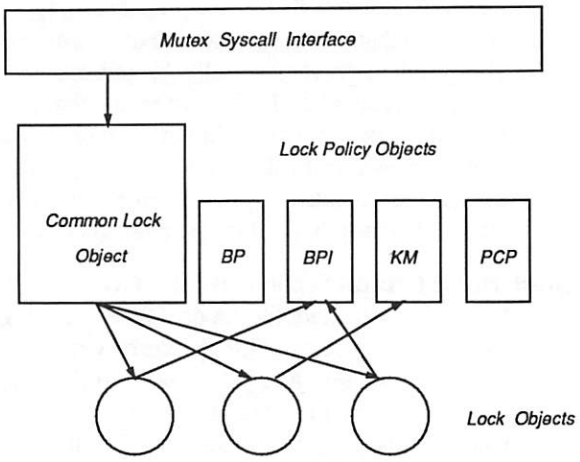


Figure 1: Lock Policy Module

manipulates the queues of threads waiting for the releasing the lock according to the current processor scheduling policy. The lock object up calls its associated lock policy object. Some of lock policy objects also controls the thread's effective priority for controlling a priority inheritance scheme.

3.2.1 Interface for Policy Control

The policy operations for the lock object are triggered by a system call from a user. The thread executes these operations without blocking. If the thread must be blocked inside these operations, it returns to the common lock object. The common lock object determines whether to block the thread or not, and re-execute the policy operations.

The lock object and policy object provide the following operations.

```
kval_t = rt_mutex_policy_acquired(lock)
kval_t = rt_mutex_policy_notacquired(lock)
kval_t = rt_mutex_policy_conflict(lock)
kval_t = rt_mutex_policy_unlock(lock, lock_attr)
kval_t = rt_mutex_policy_abort(lock, mode, who)
kval_t = rt_mutex_policy_control(lock, cmd, arg, argsize)
```

rt_mutex_lock_acquired is called when the mechanism layer needs to acquire the lock. This operation keeps track the current thread's priority for the lock object. *rt_mutex_lock_not_acquired* is used when the mechanism module cannot acquire the lock. In the case of basic priority inheritance protocol, the operation inherits the higher thread's priority. *rt_mutex_lock_unlock* is invoked from *rt_mutex_unlock*. It resets the lock structure. In the case of priority inheritance, the operation recovers the priority of the thread which executes *unlock*. *rt_mutex_conflict* is invoked when *lock* is called. It checks whether we need to abort the current critical section. *rt_mutex_lock_abort* is used when the lock is aborted. The operation is called in two ways: one

is when *lock* or *unlock* operation fails and the other is when the thread in the locked region need to be aborted. The operation is also called when timeout occurred. *rt_mutex_control* is invoked to control policy module such as set and get default lock policy.

3.2.2 Policy Module

A lock policy module is implemented as an object similar to the scheduling policy object [15]. The following lock policy objects are supported.

Kernelized Monitor (KM): No preemption is allowed while a thread is in the kernelized critical section. The duration of the priority inversion is bounded by the size of the critical region. *rt_mutex_policy_aquired* notifies the processor scheduler for entering the kernelized monitor and *rt_mutex_policy_unlock* tells the thread to exit from the kernelized critical section.

Basic Priority (BP): All operations of this policy object are null functions. The waiting threads are enqueued in the lock object based on the thread's priority.

Basic Priority Inheritance (BPI): The lower priority thread executing the critical section inherits the priority of higher priority thread, when the lock is conflicted. Priority inheritance is processed in *rt_mutex_policy_noacquired* and *rt_mutex_policy_unlock* recovers the inherited priority to the thread's original priority.

Priority Ceiling Protocol (PCP): The execution of thread is blocked when the priority ceiling of the lock is not higher than all locks which are owned by other threads. The protocol prevents deadlock, and chained blocking. The policy module has a global queue to hold the currently acquired locks. *rt_mutex_policy_acquired* checks the ceiling priority of locks in the global lock queues with the thread which will enter the critical section to determine the ceiling block of the thread.

Restartable Critical Section (RCS): The critical section is aborted by the higher priority threads and restarted after the higher priority thread is completed. *rt_mutex_conflict* is called in *rt_mutex_lock* and aborts the execution of the critical section of the current running thread if lower priority thread is executing the critical section.

The lock policy object is easy to define. Among all policies, the priority ceiling protocol is the largest policy module. The priority inheritance scheme is used in both the basic priority inheritance protocol and priority ceiling protocol. The code can be shared in both object. The advantage of the approach is not only the reduction of code size, but also it makes clear definition of each algorithm. For instance, priority ceiling algorithm is separated into priority inheritance management and priority ceiling management part.

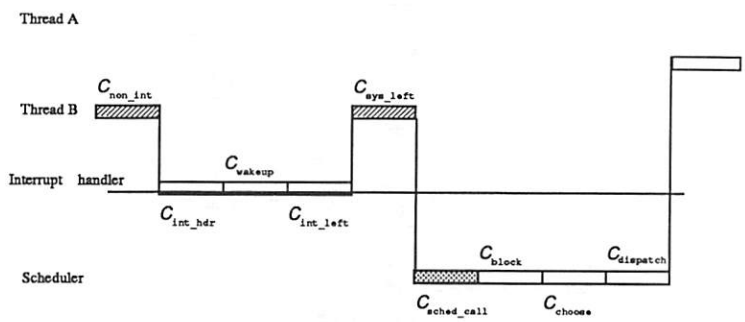


Figure 2: Preemption during a System Call

4 Evaluation and Cost Analysis

In this section, we first discuss the cost for managing real-time thread and the synchronization cost under various synchronization policies. We then analyze the relation between the schedulability and the cost of various real-time synchronization protocols.

The basic cost of the real-time thread management and synchronization primitives were measured using a Sony NEWS-1720 workstation (25 MHz MC68030) and a FORCE CPU-30 board (20 MHz MC68030). We simply evaluated a single processor environment with a Sony machine. We used an accurate clock on the FORCE board for timing measurement on a NEWS-1720 through VME-bus backplane. This clock enabled us to measure the overheads with resolution of 4 μs.

4.1 Preemption Cost Analysis

Before we start analyzing the preemptability of the system, let us first determine the basic cost factors. Figure 2 defines the basic cost factors when a higher priority thread preempts a lower priority thread which is executing a system call. C_{opr} specifies the execution cost of the primitive *opr*. C_{non_int} is the worst case execution time of a non-interrupt region where all interrupts are masked. A critical interrupt may be delayed until the non-critical region is completed. C_{int_hdr} is the worst case execution time of the interrupt handlers. Interrupt handler can be interrupted by a higher priority interrupt. C_{wakeup} is the time to wakeup a blocked thread. C_{int_left} is the remaining time after the wakeup until the interrupt is completed. C_{sys_left} is the remaining execution time of the system call. C_{sched} is the total scheduling delay time and sum of C_{sched_call}, C_{block}, C_{choose}, and $C_{dispatch}$. C_{sched_call} is the delay time to switch to the scheduler. C_{block} is the blocking time for giving up the CPU and C_{choose} is the selection time for a next thread, $C_{dispatch}$ is the context switching time. The results of the measurement are

Basic Operation	Cost (μs)
C_{wakeup}	72 †1
C_{sched_call}	72
$C_{int\_left(clock)}$	36
$C^{\dagger 1}_{block_{reincar}}$	672 †1
C_{block}	84 †1
C_{choose}	40 †1
$C_{dispatch}$	48
C_{null_trap}	48
$C_{clockint}$	108 †2

†1 C_{block}, C_{wakeup} and C_{choose}, are measured under a fixed priority scheduling policy (default) and are policy specific numbers.

†2 This includes the cost calling scheduling policy routines, but no thread wakeup cost.

†3 $C_{block_{reincar}}$ is the blocking cost at the reincarnation of a period thread.

Table 1: The Basic Overhead

summarized in Table 1.

Now, let us consider the preemption cost in Real-Time Mach. The total worst case preemption cost can be defined as

$$C_{preempt} = C_{non\_int} + C_{int\_hdr} + C_{wakeup} + C_{int\_left} + C_{sys\_left} + C_{sched}$$
$$C_{sched} = C_{sched\_call} + C_{block} + C_{choose} + C_{dispatch}$$

Under the fixed priority scheduling, $C_{preempt}$ becomes $C_{non\_int} + C_{int\_hdr} + C_{int\_left} + C_{sys\_left} + 316\ \mu s$. In our target machine, a clock interrupt handler alone requires at least additional $C_{clockint} = C_{int\_hdr} + C_{int\_left} = 108\mu s$. In a real-time application, the cost of $C_{int\_hdr} + C_{int\_left}$ can be precomputed based on the system configuration, however, the cost for C_{non_int} and C_{sys_left} are operating system specific. In many monolithic kernel-based systems, C_{non_int} and C_{sys_left} becomes relatively high. However, in a micro kernel-based system, an ordinary system call becomes preemptive since its function is implemented in a user-level thread. For real-time programs which have shorter deadlines than $C_{preempt}$, we need to reduce each cost factor further down. To reduce C_{non_int} and C_{sys_left}, further kernelization of the current micro kernel is required.

4.2 Synchronization Cost Analysis

In this section, we will analyze the synchronization cost for each locking protocol we have implemented.

In general, KM tends to be useful for a very short critical section. For medium size critical sections, we can use one of priority inheritance protocols. For very large critical sections, on the other hand, it may be better to use a restartable critical section scheme if the higher priority thread cannot afford to wait for the lower priority thread to complete its critical section.

Table 2 summarizes the execution costs of primitives, *lock and acquired*, *lock and not acquired*, and *unlock* under the five locking protocols; Basic Priority (BP), kernelized monitor (KM), basic priority inheritance (BPI), priority ceiling protocol (PCP), and restartable critical section (RCS). The *lock and acquired* cost includes the cost of trap handling, *rt_mutex_lock* in the common lock object, and *rt_mutex_policy_aquired* function in a policy object. The *lock and not acquired* cost includes only the cost for executing *rt_mutex_policy_notacquired* in a policy module. The *unlock* cost includes the cost of trap handling, *rt_mutex_unlock* function in the common lock object, and *rt_mutex_policy_unlock* function in a policy object.

Let us now calculate the cost of locking policies, and determine which protocol is better under what conditions. Here, $C^p_{nullcs(1)}$ indicates the time for a single thread to execute a pair of lock and unlock operations under a given locking protocol p and is define as $C^p_{nullcs(1)} = C^p_{syscall} + C^p_{acquired} + C^p_{unlock}$.

The measured results for KM, BPI, and PCP are as follows.

$$C^{km}_{nullcs(1)} = C^{km}_{syscall} + C^{km}_{acquired} + C^{km}_{unlock} = 288\mu s$$
$$C^{bpi}_{nullcs(1)} = C^{bpi}_{syscall} + C^{bpi}_{acquired} + C^{bpi}_{unlock} = 388\mu s$$
$$C^{pcp}_{nullcs(1)} = C^{pcp}_{syscall} + C^{pcp}_{acquired} + C^{pcp}_{unlock} = 488\mu s$$

From the above results, it was clear that the implementation cost increased proportional to the complexity of the locking protocol.

Let us then determine the worst case blocking time of the highest priority thread where n threads are sharing the same critical lock resource. The worst case for the highest priority thread may occur when a lower priority thread is already in the critical section. Then, under BPI or PCP policies, the highest priority thread becomes runnable and preempts the lower priority thread and attempts to enter the critical section. Since the resource is already in use, the highest priority thread's lock request will blocks itself and resume the lower priority thread with the highest priority (due to priority inversion). Additional overhead may occur when all other threads become runnable while the lower priority thread is executing in the critical section[1].

Under this assumption, we can determine the worst case blocking time $C^p_{wait(n)}$ for the highest priority thread under locking protocol p as follows.

$$C^{km}_{wait(n)} = C_{csbody} + (n-1) \times C_{activate} + C^{km}_{unlock}$$
$$C^{bpi}_{wait(n)} = C_{csbody} + (C_{wakeup} + C_{switch} + C_{lock\_not\_acquire} + C_{choose} + C_{dispatch}) + (n-2) \times C_{activate} + C^{bpi}_{unlock}$$
$$C^{pcp}_{wait(n)} = C_{csbody} + (C_{wakeup} + C_{switch} + C_{lock\_not\_acquire} + C_{choose} + C_{dispatch}) + (n-2) \times C_{activate} + C^{pcp}_{unlock}$$

where C_{csbody} indicates the cost for executing the body of the critical section alone, C_{switch} represents the total thread switching cost, and $C_{activate}$ is the cost for making a blocked thread runnable.

From Table 1 and 2, we can determine the cost of thread switching C_{switch} ($172\ \mu s = C_{block} + C_{choose} + C_{dispatch}$), C_{wakeup} is 72 μs, $C_{activate}$ is 108 μs, and C_{unlock} for each protocol. Then, for $n >= 2$, we can derive

[1] Assuming that the system does not disable all interrupts in KM.

	BP (μs)	KM (μs)	BPI (μ)	PCP (μs)	RCS (μs)
Lock(Acquired)	120	132	196	232	256
Lock (Not Acquired)	$208 + C_{block}$	N.A.†1	$340 + C_{block}$	$508 + C_{block}$	$628 + C_{block}$
Unlock	$144 + C_{activate} \times$ n	156/180 †3	192	256	196

†1 No one can preempt.
†2 m is a number of chains of chained locks.
†3 The cost of right parts includes the cost to wakeup the thread waiting a lock.
†4 n is a number of nest of nested locks.
†5 $C_{activate}$ is the cost for making a blocked thread runnable. The measured cost is 108 μs.

Table 2: The Cost of Locking Primitives

$$C^{km}_{wait(n)} = C_{csbody} + 180 + (n-1) \times 108\mu s$$
$$C^{bpi}_{wait(n)} = C_{csbody} + 948 + (n-2) \times 108\mu s$$
$$C^{pcp}_{wait(n)} = C_{csbody} + 1180 + (n-2) \times 108\mu s$$

From these numbers, we can conclude that as long as the system does not have any real-time thread whose deadline is shorter than $C_{csbody} + 288\mu s$ and the deadline of the real-time thread which shares the critical section is greater than $2 \times C_{csbody} + 108 \times n + 72\mu s$, then we should be able to use the KM policy among n real-time thread ($n >= 2$).

In general, the BPI and PCP protocols are suitable for a large critical section . For instance, among two synchronizing threads, the deadline of the threads should be longer than $2 \times C_{csbody} + 1336(= 948 + 388)\mu s$ for BPI. Otherwise, it may need to use KM, if there is no real-time thread whose deadline is shorter than $2 \times C_{csbody} + 288(= 216 + 72)\mu s$. Similarly for two threads under PCP, the deadline should be larger than $2 \times C_{csbody} + 1668\mu s$.

For a restartable critical section, we could get the restarting cost, $C^{rcs}_{restart}$ as follows.

$$C^{rcs}_{restart} = C_{abort} (= 816\ \mu s) + C_{recover}$$

where C_{abort} is a cost for aborting a lower priority thread and $C_{recover}$ is a cost for recovering the state of shared resource. $C_{recover}$ is application specific and there are many different schemes exit. From this restarting cost, we can also conclude that the restartable critical section is a suitable policy if $C_{restart}$ is less than $C^{bpi}_{wait(n)}$ or $C^{pcp}_{wait(n)}$.

4.3 Schedulability Test: Case I

In the following section, we will compare the schedulability of various periodic threads under different locking protocols: KM, BP, BPI, PCP, and RCS using few benchmark programs. In this Benchmark-1 program, there are four periodic threads: Th_A, Th_B, Th_C and Th_D. Th_A has the highest priority, Th_B and Th_C are medium, and Th_D is the lowest priority. Th_D is executed first, Th_B and Th_C are executed next. Lastly, Th_A is executed. Th_A and Th_D share the same object. At the beginning of the execution, each thread locks the shared object and releases the object at the end. The timing attributes of each thread are given in Table 3. C_i represents the worst case execution time of thread Th_i. B_i indicates the worst case blocking time of Th_i and T_i is the period of Th_i. U_i is the processor utilization of Th_i and S_i indicates the start time of aperiodic thread Th_i.

Thread	T_i (ms)	C_i (ms)	S_i (ms)	U_i (%)
Thread A	100	10	20	10
Thread B	300	EXE	10	$\frac{EXE}{300}$
Thread C	400	60	10	15
Thread D	1000	30	0	3

Table 3: Thread Attributes of Benchmark-1

Lock Policy	Max Exec Time (ms)	U_i (%)
BP	14	32.6
BPI	181	88.3
KM	183	89
PCP	181	88.3
RCS	202	95.3

Table 4: Breakdown Utilization under Benchmark-1

For each locking protocol, we compared the breakdown processor utilization while we vary the execution time of Th_C (i.e., EXE in Table 3). Table 4 shows the execution results of Benchmark-1. The result indicates that the priority inversion degrades the schedulability significantly and priority queueing alone does not solve the priority inversion problem. Thus, KM, BPI, PCP, or RCS is a very effective policy in real-time synchronization. Also, this benchmark result indicates that RCS policy could achieve the highest schedulability among these policies, if $C^{rcs}_{restart}$ is significantly small. Since it can abort the critical section of the lower priority thread, the higher priority thread need not to wait for the termination of the lower priority thread.

4.4 Schedulability Test: Case II

The previous benchmark shows that KM, BPI, and PCP policy may achieve relatively high schedulability for the given taskset.

Benchmark-2 presents that this is not always true under different taskset. In this program, we consider two threads, Th_A and Th_B. Th_A is the highest priority thread, and it is started 10

Thread	T_i (ms)	C_i (ms)	S_i (ms)	U_i (%)
Thread A	100	20	10	20
Thread B	200	EXE	0	$\frac{EXE}{200}$

Table 5: Thread Attributes of Benchmark-2

Lock Policy	Max Exec Time (ms)	U_i (%)
BPI	155	97.5
KM	88	64
PCP	87	63.5

Table 6: BPI, KM, PCP under Benchmark-2

ms after Th_B starts. The timing attributes of Th_A and Th_B are given in Table 5. Th_B contains a critical section, and we vary the length of the critical section during the test.

We compared BPI with KM and PCP under Benchmark-2. Table 6 summarize the breakdown utilization of each policy. Although PCP has nice properties such as deadlock free property, the result indicates that PCP could not achieve at the same level as BPI scheme. If the taskset is deadlock free in nature, then BPI policy is the best scheme.

We also compared BPI and RCS policy under Benchmark-3 program. In general, RCS becomes useful if a higher priority thread cannot afford to wait for a lower priority thread to complete the critical section. However, restarting a critical section requires not only the cost of aborting of the critical section but also the restarting cost. The schedulability may be decreased, if these costs dominates the total overhead.

Unlike Benchmark-1, Benchmark-3 indicates the case where RCS has significantly lower break down utilization than BPI.

Various benchmark programs we presented in this section indicates that there is no best policy for all real-time applications. A system designer needs to select a suitable policy for their applications.

5 Related Work

The micro kernel-based approach is gaining popularity and several micro kernel-based operating systems have been developed for an advanced distributed computing environment [4, 9, 11].

Advantages of using a micro kernel instead of a standard monolithic kernel is its high preemptability, small size, and extensibility. However, only a few micro kernels were designed for supporting predictable distributed real-time computing environment.

For instance, Chorus's micro kernel[11] was designed for real-time applications. However, their emphasis was placed at rather low-level kernel functions such as providing a user-defined interrupt handler and preemptive kernel. The kernel uses the wired-in fixed priority preemptive scheduling policy and there is no additional features reported to avoid priority inversion problems.

The V kernel's emphasis was also intended for supporting high speed real-time applications[4]. V's optimized message passing mechanism and VMTP protocol can provide basic functions for building distributed real-time applications. However, the wired-in scheduling policy and locking protocol may cause us a potential inversion problem.

Amoeba's advantage is its high performance RPC and was used for remote video image transmission using Ethernet. Like Real-Time Mach, Amoeba can support a set of single board computers without having local disks, however, it does not provide any safe mechanisms for creating a periodic thread and avoiding priority inversion problems.

The POSIX-Thread proposal[7] is very similar to Mach's C-Thread package[5]. However, it also does not distinguish between real-time threads and non-real-time threads. Like Real-Time Mach, it can dynamically select the thread scheduling policy. A POSIX thread also contains the thread attributes such as *inherit priority*, *scheduling priority*, *scheduling policy*, and *minimum stack size*. Thus, adding our timing attributes into the proposed POSIX interface would be very simple.

Uniqueness of Real-Time Mach is based on our real-time thread and synchrozation model. The proposed real-time thread model is different from many other thread models. Our model distinguishes between real-time and non real-time threads and we provide explicit timing constraints for each real-time thread. A suitable synchrozation policy such as KM, BPI, PCP, RCS can be selected to avoid unbounded priority inversion and to improve the schedulability of the given taskset.

Thread	T_i (ms)	C_i (ms)	S_i (ms)	U_i (%)
Thread A	100	20	50	20
Thread B	150	EXE	60	$\frac{EXE}{150}$
Thread C	200	60	0	30

Table 7: Thread Attributes in Benchmark-3

Lock Policy	Max Exec Time (ms)	U_i (%)
BPI	60	90
RCS	15	60

Table 8: BPI vs. RCS under Benchmark-3

6 Conclusion and Future Work

We demonstrated that using new real-time thread and synchronization facility in Real-Time Mach, a user could eliminate unbounded priority inversion problems among synchronizing threads. We also described the schedulability analysis for real-time programs in a single CPU environment. The system interface for creating periodic and aperiodic threads, creating a

mutex variable, and locking and unlocking mutex was natural and easy to specify a user policy for a specific application.

We also analyzed the performance of proposed locking protocols, KM, BP, BPI, PCP, and RCS, and determined the worst case blocking cost for real-time threads. From the analysis, we could determine which policy should be effective under what conditions.

However, these primitives alone cannot eliminate priority inversion problems in the systems. Our plan is to remodel the Mach IPC feature so that it can support a priority-based message handling and different types of transport protocols for real-time message transmission. The basic issues has been discussed and evaluated in our experimental tested [13, 14]. For instance, a priority inversion may occur when a client requests for a service to a non-preemptive server. A higher-priority client may face an unbounded priority inversion problem at the server. To avoid this problem, one solution is to add a *port attribute* for each port and allow us to set a different priority inheritance policy, like the lock attribute we implemented. Real-time network support is also important and we are planning to create a new netserver facility where protocol processing will be performed by multiple worker threads based on a message priority.

References

[1] M.J. Accetta, W. Baron, R.V. Bolosky, D.B. Golub, R.F. Rashid, A. Tevanian, and M.W. Young, "Mach: A new kernel foundation for unix development", *In Proceedings of the Summer Usenix Conference*,, July, 1986.

[2] Özalp Babaoglu, "Fault-Tolerant Computing Based on Mach", In *Proceedings of USENIX Mach Workshop*, October, 1990.

[3] R. Chen and T.P. Ng, "Building a Fault-Tolerant System Based on Mach", In *Proceedings of USENIX Mach Workshop*, October, 1990.

[4] D.R. Cheriton, G.R. Whitehead and E.D. Sznyter, "Binary emulation of UNIX using V Kernel", *In proceedings of Summer Usenix Conference*, June, 1990.

[5] E. C. Cooper, and R. P. Draves, "C threads", Technical report, Computer Science Department, Carnegie Mellon University, CMU-CS-88-154, March, 1987.

[6] D. Golub, R. Dean, A. Forin, and R. Rashid, "Unix as an application program", *In the proceedings of Summer Usenix Conference*, June, 1990.

[7] IEEE, "Realtime Extension for Portable Operating Systems", P1003.4/Draft6, February, 1989.

[8] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment", *Journal of the ACM*, Vol.20, No.1, 1973.

[9] S.J. Mullender, G.V. Rossum, A.S. Tanenbaum, R. Renesse and H. Staveren, "Amoeba: A Distributed Operating System for the 1990s", *IEEE Computer* Vol.23, No.5, May, 1990

[10] R. Rajkumar, "Task Synchronization in Real-Time Systems", Ph.D. Dissertation, Carnegie Mellon University, August, 1989.

[11] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemount, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser, "Chorus distributed operating system", *Computing Systems Journal*, The Usenix Association, December, 1988

[12] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization", Technical Report CMU-CS-87-181, Carnegie Mellon University, November 1987

[13] H. Tokuda and C. W. Mercer, "ARTS: A distributed real-time kernel", *ACM Operating Systems Review*, Vol.23, No.3, July, 1989.

[14] H. Tokuda, C. W. Mercer, Y. Ishikawa, and T. E. Marchok, "Priority inversions in real-time communication", In *Proceedings of 10th IEEE Real-Time Systems Symposium*, December, 1989.

[15] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time Mach: Towards a Predictable Real-Time System", In *Proceedings of USENIX Mach Workshop*, October, 1990.

How to Design Reliable Servers using Fault Tolerant Micro-Kernel Mechanisms

Michel Banâtre Pack Heng◊ Gilles Muller* Bruno Rochat◊*

** IRISA/INRIA*
◊ BULL Research
IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
e-mail : ftm@irisa.fr

Abstract

The purpose of the Fault Tolerant Multiprocessor (FTM) project is to design a fault tolerant machine based on Stable Transactional Memories (STM). The FTM operating system is built from a MACH/OSF kernel which is extended to provide reliable services.

The purpose of this paper is to describe the basic mechanisms that we have added to the MACH micro-kernel in order to achieve fault tolerance.

1 Introduction

The Fault Tolerant Multiprocessor FTM [3] is a general purpose fault tolerant machine based on the association of Stable Transactional Memory (STM) boards with standard open multiprocessor machines. The STM is a fast stable storage device providing atomic memory accesses with low time overhead compared to normal RAMs. The FTM architecture can tolerate any *single hardware fault.*

Unlike STRATUS [8] and TANDEM S2 [9] architectures which mask hardware faults using static redundancy, the FTM architecture is based on dynamic redundancy, e.g. all the processors are performing different jobs in normal operating mode. In the event of a failure, the task of the faulty processor is handled by a backup one in addition to its own jobs. Thus the operating system has to deal with hardware faults and mask them from the users.

Our goal in the FTM operating system is to help the design of *reliable service*s which mask hardware faults from its clients. A reliable service is composed of one or more reliable servers; each reliable server is implemented by storing its variables in STM and by other fault tolerant mechanisms that we propose to add to the MACH 3.0 micro-kernel [1].

The following is structured as follows. In section 2, we present the FTM architecture along with the STM functionalities. In section 3, we describe the basic mechanisms that we have added to the MACH kernel to support reliable servers. In section 4, we give an example of a reliable server running on the extended micro-kernel. We conclude in section 5.

2 The FTM Architecture

The FTM architecture can be seen as a virtual loosely-coupled multiprocessor, in which the processing element is the *stable node.* The FTM architecture ensures the following three properties:

This research was supported in part by the DRET under the grant n° 90346.

1. The architecture tolerates any single hardware fault.
2. A stable node can restore a safe state of data after an internal failure and is able to resume computations.
3. The interconnection medium ensures that communication is always possible between two stable nodes and that no partition even occurs.

2.1 The Stable Node

A stable node S_{ab} (see Figure 1) is built from a primary processor P_a, a backup processor P_b and a Stable Transactional Memory STM_{ab}. The processor P_a can manipulate locally the stable variables in STM_{ab} with a fast access time. In normal operating mode, the STM_{ab} is not shared between P_a and P_b.

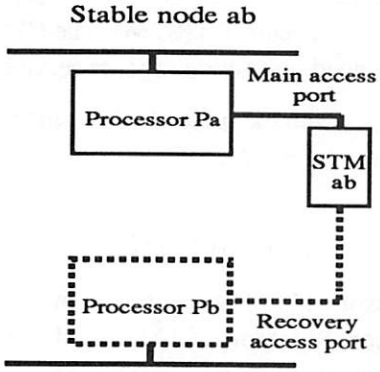


Figure 1. The stable node

The STM_{ab} is able to detect P_a processor failures using a watch-dog mechanism. When the STM_{ab} detects the failure of P_a, it restores a consistent state of the stable data and warns P_b. P_b then resumes aborted computations from the stable data stored in STM_{ab}. After a manual maintenance on the faulty processor P_a, it is restarted in the stable node S_{ab}. The STM_{ab} disconnects the backup processor P_b and reconnects the primary processor P_a. The computations resume on P_a. If both processors P_a and P_b fail, the computations running on the stable node S_{ab} are stopped but can be restarted as soon as one the processor P_a or P_b restarts.

Stable nodes are physically paired in such a way that the primary processor of one stable node is the backup processor of the other stable node (see Figure 2). Thus, there is no backup processor dedicated to fault tolerance in the FTM architecture (dynamic redundancy).

The physical FTM architecture is based on open multiprocessor machines. Its complete and detailed description, particularly of the interconnection medium, can be found in [3].

2.2 The Stable Transactional Memory

The STM provides two notions: the *stable object* and the *transaction*. A stable object is a contiguous set of memory words and a transaction is an atomic set of basic operations performed on the stable objects. All STM operations *can only be performed* within transactions. Objects of any size from kernel lists of few words elements to virtual memory pages of several Kilobytes can be managed atomically.

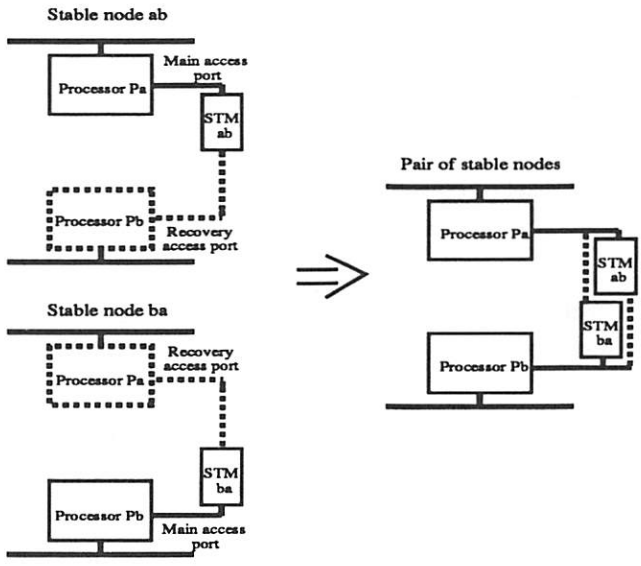


Figure 2. The pair of stable nodes

The STM is internally implemented using two banks of 32 Megabytes of battery-backup RAM memory and an intelligent controller which implements transactions using a two phase commit protocol.

2.2.1 Stable Objects and Protection

For performance purposes, the STM is directly accessed in the physical address space of a processor as a local memory. Generally, processors are not *fail-stop*: a faulty processor may thus corrupt accessible memory. To protect the STM, stable objects have to be made visible using an explicit *open* operation, before being modified by the processor. Similarly, after modification the object have to be closed using a *close* operation. Any access to a non opened or non existent object is detected and rejected by the STM.

The processing of an STM access violation by the micro-kernel depends on the level of the faulty software. If the fault comes from the kernel or a well tested server, we assume that there is some hardware malfunction and the processor is halted. If the fault comes from an application software or a server currently being tested, the faulty program is aborted.

2.2.2 The Transaction Facility

The transaction is the most important functionality of the STM. It allows the programming of complex atomic functions. The operations on transactions are *Begin*, *Commit*, *Abort*. When a transaction is committed (resp. aborted), all effects on stable objects are validated (resp. invalidated).

Several transactions can be used concurrently at any time. Thus a transaction is identified by a descriptor stored itself in the STM. The creation of a new transaction descriptor is performed atomically within another transaction and is distinct from its activation (*Begin*). When the STM detects a processor failure, it automatically aborts active transactions.

3 Basic Fault-Tolerant Mechanisms of the Micro-Kernel

3.1 Overview of the MACH Kernel

Recent evolution in the design of operating systems has advanced the notion of *micro-kernel.* A micro-kernel offers a minimal set of concepts allowing the implementation of complex distributed systems and applications in a modular way. MACH [1] and CHORUS [12] are examples of well known micro-kernels.

There are five basic entities in MACH: task, thread, port, message and memory object. A task is an allocation context of resources: it possesses a logical address space and access rights to ports or memory objects.

A thread is the execution unit and is attached to some task which defines its execution environment. Several threads can be created within the same task and share the same resources. In particular, they share the same address space and can communicate directly by shared memory.

A port is the communication resource and allows different threads to exchange messages. Several tasks can possess the send right to a port, but only one has the receive right. Rights can be transmitted by messages but only under the control of the kernel. Ports also permit to identify any MACH object.

A memory object must be mapped into a task's address space before being accessed by a thread. If a memory access raises a page fault, the kernel sends a message to the memory object port notifying it of the fault. The page fault can be treated by an external server (pager) which owns the page and sends it back to the kernel.

The five MACH basic entities allow to implement servers. A server is a task which performs a cyclic job. It receives requests from clients, treats them and returns results. Requests can be served by one or several threads. Clients are identified by their sending port and the server by its receipt port.

3.2 The FTM Fault Tolerant Micro-Kernel

The FTM micro-kernel extends MACH by adding mechanisms which allow the implementation of fault-tolerance. Normal MACH entities, as tasks, ports, memory objects, can be corrupted by a processor failure. To restore a safe system state, the solution is to create stable equivalent MACH entities. The FTM micro-kernel provides the standard MACH entities and in addition three other ones: *stable task*, *stable port* and *stable memory object.* All these stable entities are created and modified within STM transactions. Transactions are executed by threads and ensure consistency of stable entities in case of a processor failure.

3.2.1 The Stable Task

A stable task defines an execution environment which is associated with a stable node. As with a normal task, a stable task contains access rights and a logical address space. Each time a processor switch occurs, the environment is restored on the active processor by the FTM micro-kernel. Then a thread is restarted from the entry point of the task.

3.2.2 The Stable Memory Object

A stable memory object can be viewed as a logical STM. It has the same functionalities as the physical STM described in 2.2. A stable memory object is mapped into a stable task and is private to this task. Thus, threads can activate transactions, create and modify stable objects.

The FTM micro-kernel manages the sharing of the physical STM between the different stable memory objects. In particular, it performs STM page allocation and deallocation.

3.2.3 The Stable Port

A stable port is used to perform reliable communications between two threads. Like any of the operation defined on stable entities, operations on stable ports are performed within transactions. That means that sending or receiving messages is effective only at the transaction commitment.

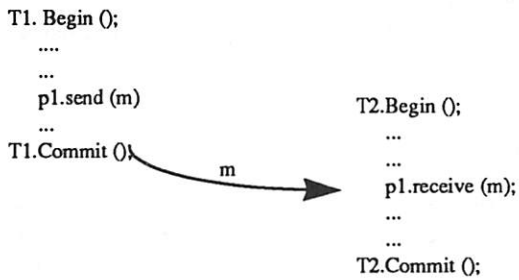


Figure 3. Message exchange between two tasks

For instance, in Figure 3, the message *m* is only sent when *T1* commits. Similarly, if *T2* aborts, the message is put back in the receipt queue of *p1*.

Stable ports are implemented in the micro-kernel, using lists of messages which are stored in STM. A reliable network message server is used when threads are located on different stable nodes. One of its functions is to locate the active processor of the destination stable node.

3.3 The FTM Reliable Server

A reliable server masks hardware faults from its clients and is implemented using a stable task. In the event of a processor failure, the stable task is restored on the active processor of the stable node and a thread is restarted on the entry point. Each server operation is executed atomically within an STM transaction. If the server fails, the message is put back in the port and the modified stable objects are restored to their original state. Thus, the atomicity property in the treatment of each remote procedure call ensures that the server does not lose any client message and executes *exactly once* each call.

As a normal RAM memory, the STM is shared between reliable servers. Its 32 Megabyte capacity satisfies all system needs. User applications, which are designed for the FTM, may also have access to STM. However, to support non fault tolerant applications, we have designed a reliable distributed virtual memory server [2] which allows to run transparently users programs. This server integrates in a uniform view all physical memories of the system (RAM, STM and disks).

4 Programming Example

4.1 Programming with the STM

A C++ interface of the STM has been designed to help the programming of applications. Two classes, *Transaction* and *Stable* are defined in this interface and mask the STM hardware to the C++ programmer. Moreover, the interface allows the definition of a C++ stable object simply by inheritance of the *Stable* class [11]. A similar inheritance mechanism is also proposed in Arjuna [6] and Avalon [5].

The C++ STM interface is defined as follows:

```
class Stable {
public:
   operator new (long s);                  /* Creation of new stable object of size s */
   operator delete (void *pt);             /* Destruction of the object pointed by pt */
   void Open ();                           /* Open the stable object */
   void Close ();                          /* Close the stable object */
};
class Transaction {
public:
   operator new (long s);                  /* Creation of a new transaction descriptor */
   operator delete (void *pt);             /* Destruction of the transaction descriptor */
   void Begin ();                          /* Activation of the transaction */
   void Commit ();                         /* Commit the transaction */
   void Abort ();                          /* Abort the transaction */
};
```

To illustrate programming using the STM, we treat the example of stable segment management in a virtual memory. We first define a *Segment* class independently of the STM mechanisms. To allow all stable segments to be allocated in the STM, the *Segment* class inherits from the *Stable* class. We present in the following example a simple segment class with two operations which read and write a page (the page has a fixed size of 1 Kilo bytes).

```
const int page_size = 1024;
typedef char[page_size] Page;                          /* page type */
class Segment : public Stable {
   const int segment_size = 10*page_size;              /* 10 K bytes */
   char S[segment_size];          /* the segment is implemented by an array of characters */
public:
   void read_page (int no_page, Page& page);           /* read a page */
   { int offset = (no_page*page_size)%segment_size;    /* offset in the segment */
     for (i=0; i<page_size; i++) page[i] = S[i+offset];
   }
   void write_page(int no_page, Page page)             /* write a page */
   { int offset = (no_page*page_size)%segment_size;    /* offset in the segment */
     for (i=0; i<page_size; i++) S[i+offset]=page[i];
   }
};
```

As shown on the previous example, the programmer of a stable class does not need to know the STM mechanisms. On the contrary, the user of a stable segment has to manage explicitly transaction and object visibility. This is shown in the following example:

```
Transaction T;                  /* transaction which manipulates a stable segment */
Segment *s;                     /* pointer to a stable segment */
Page page;                      /* a volatile page */
```

```
/* atomic creation of the stable segment */
T.Begin (); s = new Segment (); T.Commit ();
T.Begin ();                                  /* atomic move of page 0 to page 1*/
  s->Open ();
    s->read_page (0, page);
    s->write_page (1, page);
  s->Close ();
T.Commit ();
```

4.2 Example of a Reliable Server

We are now extend the previous example by designing a reliable server which offers in its interface, the *read*_page and *write*_page operations on a segment. The clients call these operations by remote procedure call.

A request message sent by a client to the server contains:

- the receiver stable port to which the message is sent,
- the sender stable port from which a result is eventually waited for,
- the type of the request (READ_PAGE, WRITE_PAGE, RESULT),
- the page number of the page which is read or written.

The client which calls a remote operation atomically sends a message to the server stable port and atomically waits for a result. The following example describes the remote procedure call to *write_page (0, page)*:

```
typedef unsigned int StablePort;          /*stable port identifier */
enum Request_t {READ_PAGE, WRITE_PAGE, RESULT};
struct {
  StablePort receiver_port;
  StablePort sender_port;
  Resquest_t request;
  int page_number;
  Page page;
} Message;

Transaction TClient;                      /* transaction of the client */
StablePort client_port;                   /* stable port of the client */
Page page;                                /* the page to write */
TClient.Begin ();                         /* Atomic send of the write_page request */
 server_port.send (
  Message client_message=(server_port, client_port, WRITE_PAGE, 0, page));
TClient.Commit ();
TClient.Begin ();                         /* atomic wait for a result */
  client_port.receive (Message result_message);
TClient.Commit ();
```

As a message is only sent at the commitment of the transaction. It is necessary that *Tclient* commits between the send and receive operations. Otherwise the client would be indefinitely blocked on the receive operation.

In the treatment of a client request, the server atomically receives the message sent by the client, performs the operation on the stable segment and sends back the result.

```
Transaction TServer;                      /* transaction of the server*/
StablePort server_port;                   /* stable port of the server */
Segment s;                                /* stable segment */
```

```
main()
{ /* infinite loop */
 while (TRUE) {
   Message client_message;              /* message received from a client */
   Page result_page = "";               /* result page */
   /* Atomic transaction : receive a message, treats an operation, sends a result */
   TServer.Begin ();
   server_port.receive (client_message);
   switch (client_message.request) {
   case WRITE_PAGE:
     s.Open ();
     s.write_page (client_message.page_number, client_message.page);
     s.Close ();
     break;
   case READ_PAGE:
     s.Open ();
     s.read_page (client_message.page_number, result_page);
     s.Close ();
     break;
   }
   client_message.sender_port.send (Message message_result =
     {client_message.sender_port, server_port, RESULT,
     client_message.page_number, result_page});
   TServer.Commit();
 }
}
```

5 Discussion

Using atomic actions to build robust distributed programs has been extensively investigated in ARGUS [10] and CAMELOT [7]. FTM contributes to this work by providing an efficient implementation of atomic actions on small data objects (arrays, lists). Thus, transactions can be used within the design of kernel and system services. Generally the implementation of remote operations only offers the semantics *at most once*. Using stable ports, we can offer the *exactly once* semantics.

As mentioned in this paper, the failure of a reliable server is transparent to the clients. However, if the client fails, its effects on the server are not undone. The coordination between client and server needs other mechanisms as distributed atomic actions. We are now working on the implementation of atomic action which are dynamically determined using communication between clients and servers.

The STM boards are currently under development. Consequently we are prototyping the FTM micro-kernel using a SUN 3 version of MACH 3.0 and a C++ emulation of the STM. In parallel, we are also porting MACH 3.0 on to the Motorola 68030 based processors of the FTM.

References

[1] M. Accetta and R. Baron and W. Bolosky and D. Golub and R. Rashid. *A New Kernel Foundation for UNIX Development. USENIX 86*, July 1986.

[2] M. Banâtre, G. Muller, B. Rochat, P. Sanchez. A Reliable Distributed Virtual Memory on top of the Mach kernel. *OSF Micro Kernel Applications Workshop*, Grenoble, France, November 1990.

[3] M. Banâtre, G. Muller, B. Rochat, and P. Sanchez. Design Decisions for the FTM : A General Purpose Fault Tolerant Machine. In *Proc. of 21th International Symposium on Fault-Tolerant Computing*

Systems, pages 71-78, Montréal, Canada, June 1991.

[4] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of Synchronization and Recovery Properties in Avalon/C++. *IEEE Computer*, pages 57–69, December 1988.

[5] G.N. Dixon and S.K. Shrivastava. Exploiting Type Inheritance Facilities to Implement Recoverability in Object Based Systems. In *Proc. of the 6th Symposium on Reliability in Distributed Software and Database Systems*, pages 107–114, Williamsburg, March 1987.

[6] J. L. Eppinger and A. Z. Spector. A Camelot Perspective. *UNIX REVIEW*, 7(1):58, 1989.

[7] E. S. Harrison and E. Schmitt. The Structure of SYSTEM/88, a Fault-Tolerant Computer. *IBM Systems Journal*, 26(3):293–318, 1987.

[8] D. Jewett. Integrity S2: A Fault-Tolerant Unix Platform. In *Proc. of 21th International Symposium on Fault-Tolerant Computing Systems*, pages 512–519, Montréal, Canada, June 1991.

[9] B. Liskov and R. Scheifler. Guardians and actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[10] G. Muller, B. Rochat, and P. Sanchez. A Stable Transactional Memory for Building Robust Object Oriented Programs. In *EuroMicro 91*, Viennes, Autriche, September 1991. to appear.

[11] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, P. Léonard, S. Langlois, and W. Neuhauser. The Chorus Distributed Operating System. *Computing Systems*, 1(4), 1988.

The File System Belongs in the Kernel

Brent Welch
welch@parc.xerox.com
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

Abstract

This paper argues that a shared, distributed name space and I/O interface should be implemented inside the operating system kernel. The grounding for the argument is a comparison between the Sprite network operating system and the Mach microkernel. Sprite optimizes the common case of file and device access, both local and remote, by providing a kernel-level implementation. Sprite also allows for user-level extensibility by letting a user-level process implement the naming and I/O interfaces of the file system. Mach, in contrast, provide general interprocess communication and does not define a file system protocol in the kernel.

1 Introduction

This paper argues that the file system is a mature enough abstraction that it should be implemented inside the operating system kernel for optimal performance. Data storage and high-level naming are fundamental features of today's computer systems, along with process and communication abstractions. In the Sprite network operating system, a shared network file system is the basis for a distributed system, and it is implemented inside the operating system kernel. The kernel provides the framework for a transparently distributed hierarchical name space and for an I/O stream abstraction with UNIX semantics. The Sprite file system provides higher performance than NFS and AFS [Nelson88] [Ousterhout90]. Unlike these two systems, the Sprite file system retains the full semantics of the UNIX I/O stream abstraction even in cases of network sharing [Welch90].

In comparison, the central abstractions in Mach are ports, messages, tasks, threads, and memory objects. These are lower-level, more general purpose abstractions than those of a file system. Therefore, many would argue that the operating system should provide only these general-purpose facilities and defer higher-level facilities like the file system to implementation at user-level. However, I argue that our experiences with UNIX have taught us the value of the file system abstractions, so they merit efficient support in a kernel-level implementation. Consider the following quote from Butler Lampson's article *Hints on Computer System Design* [Lampson83].

> "If the interface is used widely enough, the effort put into designing and tuning the implementation can pay off many times over. But do this only for an interface whose importance is already known from existing uses. And be sure that you know how to make it fast."

The file system satisfies both of Lampson's criteria; its importance is well-known and there are caching techniques that make it perform well. The fundamental differences between the VM interface and the file system interface will discussed in more detail below.

Aiming at a high-level interface has two benefits. First, the interface provides more functionality so its clients are simpler. In particular, the Sprite file system abstraction handles network transparency. Second, in focusing on a high-level interface, the designers have more freedom to make different design decisions. Sprite optimizes network access by providing a kernel-to-kernel RPC protocol, and it optimizes remote file access by providing a distributed caching scheme. It optimizes performance in general by putting the implementation inside the kernel. The hierarchical file system name space has names for a variety of object types, not just files.This requires careful design of the file system interfaces, which are described in more detail below. The final touch is to add an upcall facility[Clark85] so that a user-level process can implement the naming and I/O interface for functions we did not wish to put into the kernel. Thus Sprite takes a hybrid approach. It chose a well-known, high-level, widely used interface and supports it in the kernel, yet provides an escape hatch to user-level for more arbitrary functionality.

In contrast, a microkernel requires an external file system implementation in order to be generally useful. The strategy taken by Mach is to layer file access on top of the virtual memory system by using mapped files. The section below describes why this is not necessarily a good idea, especially in a network environment. Also, one of the nice features of the UNIX file system is the hierarchical name space. The only names implemented in the Mach microkernel are ports, which are too low-level to be useful by themselves. Instead, a name service is also required for the microkernel to be useful. In short, you'll need a file system anyway, and layering it on top of a virtual memory-oriented, message-passing kernel is not the best way to go about it. The next few sections elaborate on this point.

2 The File System vs. Virtual Memory

The fundamental difference between the virtual memory interface and the file system interface is that the virtual memory interface is used on every load and store instruction, while the file system interface is invoked at a few well defined moments. As a result, considerable hardware support is provided for the VM system so it performs well. Translations from virtual to physical addresses are cached in fast registers, and CPUs are designed so they can make virtual memory references with little or no penalty. However, this model does not extend well to networks of loosely coupled machines. Work by Kai Li has demonstrated that it is feasible, but his work is applicable to a certain class of applications that share memory at a relatively infrequent rate [Li89].

In contrast, the file system is oriented towards access to slow devices such as disks and terminals. In these cases, it is more natural to use read and write procedures that move data from the slower devices into main memory. Open and close procedures provide convenient points at which the system can set up state about the underlying device in order to optimize later read and write calls. In particular, setup can be performed to optimize network accesses. Sharing resources is also easier via the file system interface because the calls into the file system interface provide well defined points at which to check for consistency among shared copies. In contrast, network shared memory schemes must rely on complex page protection schemes that grant ownership of a page to a particular process.

Consider the hard case for network sharing when processes on different machines are sharing and modifying the same page of data. If the page is shared via the VM interface, the system must arrange page protections so that one process has read-write privileges, while

all other processes have no access rights. The later processes will suffer a page fault and the cost of copying a whole hardware pageframe across the network when they try to access the shared page. Recall that this can occur as often as every load and store instruction. In contrast, the Sprite file system uses a simple trick to handle the hard case. It simply disables the caching for the shared file, forcing the read and write calls to go over the network to the server for the file[Nelson88]. I/O operations are serialized in the main memory file cache of the server. In most cases less data is transferred over the network, too, because only the modified bytes need to be sent to the server in this case, not the whole page. This example illustrates one of the fundamental differences between the file system and VM. A virtual memory page always has to be resident in physical memory in order to access it, while file system data can be streamed into memory from the device, often using a highly tuned hardware channel interface. Granted, the file system interface implies a copy operation from the device into the user's memory, but this occurs when the programmer invokes read, plus it can be further optimized by caching the data in main memory.

Another difference between the two systems is the way they access memory. Accesses to file system data is by and large sequential, while virtual memory accesses are more randomized. Systems that layer the file system on top of mapped files often suffer performance problems because of this. The classic bug is that copying a large file can throw out the working set of the running programs, slowing down all those load and store instructions. In Sprite, the file system and the virtual memory system each maintain their own pool of memory pages. When they run out, they negotiate over who should give up a page. The two systems compare their oldest LRU times, and system with the oldest time gives up a page. This technique is refined by biasing against the file system. The VM system cheats and adds 20 minutes to its time before telling the file system. This means that no VM page that has been accessed within the last 20 minutes will be thrown out in favor of a file system cache page [Nelson88].

3 Micro vs. Monolithic

"So what?", you say, given efficient IPC, you could perform all those tricks in the file system that accompanies the microkernel. However, the microkernel makes a fundamental trade-off between security clearance procedures and performance. Smaller programs are easier to certify as secure, so the goal of the Mach microkernel is to have a small kernel and a set of small system servers. However, by moving system services out of the kernel address space there are inevitable performance consequences.

First, the resources controlled by the file system are ultimately accessed via kernel-resident device drivers. Note that in Sprite, this includes access to the network as well as to disks and other peripheral devices. Therefore, the execution path ultimately has to end up inside the kernel. If the file system is implemented in a separate address space, then file system accesses suffer additional traps into the kernel in order to effect the change of address space. Furthermore, each additional address space that is crossed places more load on the MMU. For example, the Sun MMU is designed to support efficient access to 8 or 16 address spaces, one of which is reserved for the kernel. The DECstation has a 64 entry TLB, but it can get away with this because most of the operating system kernel resides in a memory range that is mapped one-for-one to physical pages, thus bypassing the small TLB. Processor caches have similar problems. Often a change of address space requires a cache flush. In the Suns, the cache is only flushed when one of the hardware contexts is reused, but this

will happen more often when there are more address spaces placed in the critical path of the file system. Finally, each address space crossing adds more instructions to the execution path. There is an inevitable amount of glue code, however small, associated with the boundary. If the message abstraction is added, then there is even more code to pack and unpack things from the messages.

Thread scheduling is another potential source of overhead in the microkernel approach. Not only does the execution path cross address spaces, but a new thread must be found to execute in the other address space. This overhead is not present in a kernel-resident system. The application thread traps into the kernel and continues to execute in the kernel's protected address space. After that, communication between various kernel-provided services is achieved with a simple procedure call.

Recently, Bershad has extended this idea to apply to user-level IPC, eliminating the need to schedule a new thread [Bershad90]. In Bershad's LRPC mechanism an execution stack is mapped into a server process for each client that is bound to it. The client thread traps into the kernel when crossing the protection boundary, but it continues to execute on the shared execution stack after the kernel fixes up the address space. Using this technique, Bershad reports a null round-trip time of 125 microseconds on a 3 MIP C-vax. In contrast, Draves reports the same time for an optimized version of Mach IPC, but on a 12 MIP DECstation [Draves90].

In spite of recent advances in IPC by Bershad, which I applaud, the fact remains that execution paths will be longer in services that are provided outside the operating system kernel. Extra glue code at the interfaces, traps into the kernel, and MMU effects from changing address spaces all contribute to overhead. Given that the file system is as fundamentally important as the VM interface, and the potential problems with implementing one in terms of the other, it seems reasonable to provide an efficient, kernel-level implementation of the file system.

4 Benefits of a Shared File System

A UNIX file system supports two main abstractions, pathnames and I/O streams. These abstractions were derived from earlier work in Multics [Feirtag71]. Experiences with these abstractions have shown that the notions of device-independent naming and I/O are extremely useful, and that the lack of them in a network environment is frustrating. Accordingly, Sprite extends these file system abstractions to a network environment. Additionally, Sprite provides process migration so that cycles can be shared across the network. The combination of a shared file system and process migration makes a network of Sprite workstations into a powerful computing platform.

If the file system is chosen as the basis for the system, a number of simplifications are possible. First, the file system can act as the name space for the system. UNIX, for example, uses special files to represent peripheral devices. Additionally, Sprite uses special files to represent user-level server processes known as *pseudo-devices* [Welch88]. The services implemented as pseudo-devices include a TCP/IP protocol server, terminal emulators, and the X display server. More details about the pseudo-device mechanism will be given below.

Another simplification possible in Sprite is that regular files are used as virtual memory backing store as opposed to having preallocated, dedicated swap space. This is especially convenient in a network of diskless workstations. First, it is not necessary to preallocate

swap space on disk as it is in most UNIX systems. Second, a remote file server can share a swap directory among many clients. This approach is valuable in today's networks of workstations with large memories and applications with large working sets. The Sprite network at Berkeley uses a single, 600 Meg disk for the backing store of over 40 hosts.

The shared file system also simplifies the implementation of process migration. An address space is moved to a new host simply by paging it all out to the shared file system and demand paging it back it at the new site. Similarly, there is no difficulty with data files or device access after a migration because all file system resources are uniformly available on all hosts. I will admit, however, that the algorithm to correctly migrate an open I/O stream while preserving the semantics of shared UNIX I/O streams was tricky to get right.

5 Sprite Features and Development History

Sprite is a 4.3 BSD UNIX compatible operating system with extensions for a distributed file system, process migration, multi-threaded address spaces, and a multi-threaded kernel for use on a multiprocessor. The kernel was coded from scratch in C, from the device drivers and boot code up through the system call layer. The project began with professor John Ousterhout and 4 graduate students: myself, Andrew Cherenson, Fred Douglis, and Mike Nelson. Later students included Mendel Rosenblum, Mary Baker, John Hartmann, Ken Sherriff, and Jim Mott-Smith. Adam de Boor, Robert Bruce, and Mike Kupfer were valuable staff members.

The facilities implemented in the Sprite kernel include:

- A debugnub to support remote kernel debugging.
- Device drivers.
- A kernel-to-kernel RPC network protocol.
- Address spaces with virtual memory.
- Multiple threads of execution per address space.
- A transparently distributed hierarchical name space.
- An transparently distributed I/O interface.
- A local file system.
- Host monitoring and failure recovery.
- Integration of user-level services into the name space and I/O interfaces.
- Process migration.

The features listed above are given in the rough order they were implemented, although there was considerable overlap. Before the "official" start of Sprite, I had modified SunOS 1.1 to use prefix tables to distribute the name space uniformly among workstations [Welch86b], and to use a kernel-to-kernel RPC protocol for network communication [Welch 86a]. Implementation of the true Sprite kernel began in the Summer of 1985 using Sun2 workstations. The debugnub was built early so we could use a symbolic debugger via a remote workstation running UNIX. File service was initially provided by the prototype Sprite file server. This approach let us defer writing the first local Sprite file system, including disk drivers and a file system format, until the Summer of 1986. Work on the distributed caching system followed soon after the native file server was up. In the fall of 1987 the system sources were moved to a native Sprite file server, a Sun3 with 16 Meg of main memory, and all development continued using Sprite itself. Work continued on graceful failure recovery, user-level extensions, and process migration. During the '87-88 academic year I

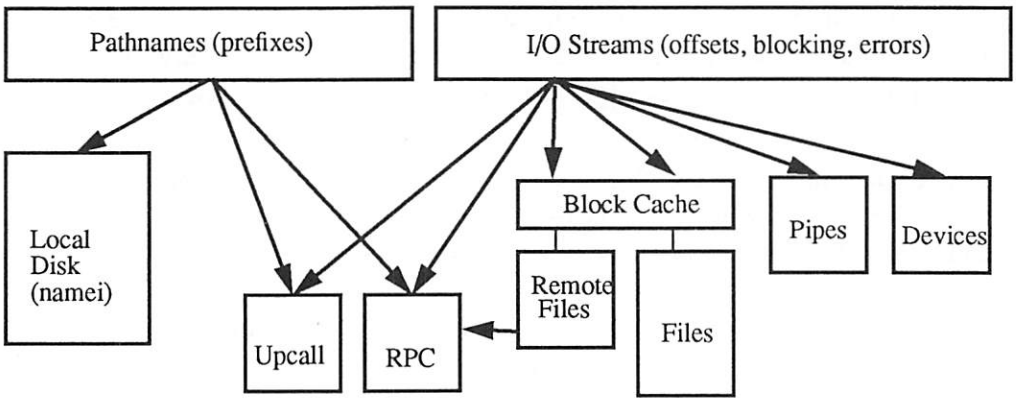


Figure 1. An overview of the Sprite file system architecture. The two primary interfaces involve pathnames and I/O streams. The Upcall module forwards operations to user-level processes. RPC forwards operations across the network to other Sprite kernels.

redesigned and reimplemented the file system architecture to better support the many features that had crept into the system. In 1989, Sprite was ported to the Sun4, DECstation, and SPUR multiprocessor [Hill86]. In the Fall of 1989 the Sprite network expanded to support a number of regular users that included professors and grad students working on other projects. The most recent work on Sprite is the log-structured file system[Rosenblum90][-Rosenblum91] and extensive measurements of the system's day-to-day performance [Baker91]. Today the main Sprite network at Berkeley includes 1 primary file server, a Sun4 with 128 Meg of main memory, 3 auxiliary file servers, and about 40 workstations. The time from initial development to day-to-day usage by non-developers was about 5 years, which matches experiences with other operating system projects [Lauer81].

In contrast, Mach began as a modification of a 4.2 BSD kernel. This gave it a nice head start, and it was quickly used on a number of multiprocessors because it improved on the BSD virtual memory interface that was oriented heavily towards the VAX architecture. In 1986, a USENIX paper hailed Mach as a "new kernel foundation for UNIX development." [Accetta86] However, 5 years later, the microkernel and its accompanying set of server processes is still under development. My conclusion is that it is ultimately cleaner to start from scratch, although there is an initial start-up penalty as basic features are reimplemented.

6 Sprite File System Architecture

The internals of the file system were rewritten in an object-oriented style during the period of about one year (academic year '87-88) to clean up and simplify the interactions among various features. The redesign introduced a generalized *object descriptor.* An object descriptor is a main-memory data structure maintained by the kernel, not a disk-resident representation of an object. A basic object descriptor has a type, uid, server ID (a sprite Host ID), a reference count, and a lock bit. Objects are specified internally by a tuple of <type, serverID, uid>. This base data structure is subclassed* for the implementations of various object types. The object-oriented approach allowed clean separation of different object

*. The various kinds of object descriptors would be the result of subclassing if Sprite were written in C++. However, all the object-oriented features described here were hand crafted in C.

implementations, as well as some sharing. At the same time, the interface between the top-level, generic layers of the file system and the lower-level, object-specific layers was cleaned up based on experiences with the initial design. A diagram of the file system architecture is given in Figure 1.

The pathname interface illustrates the three basic cases handled by the Sprite kernel. The server for a pathname may be the local kernel, in which case the file system implementation is accessed by an ordinary procedure call within the Sprite kernel. The server may be remote, in which case the kernel-to-kernel RPC protocol is used to pass the pathname to the server. Finally, the server may be a user-level process, in which case an upcall mechanism, which is described below, is used to pass the operation up to user level. Thus there are three orthogonal cases that are supported by a Sprite kernel, a local, kernel-resident module, a remote module, and a user-level module.

The importance of this approach is that it extends the general features implemented in upper levels of the kernel to local, remote, and user-provided objects. Notable, high-level features include the name space, error recovery, and blocking I/O. Thus, the focus of the file system architecture has relatively little to do with actual disk management. The focus is on extending the high-level abstractions of pathnames and I/O streams to the network environment.

7 The Sprite Name Space

Sprite uses a prefix table mechanism to implement a uniformly shared, hierarchical name space. Each Sprite kernel keeps a cache of pathname prefixes. The prefixes define the way server domains are coalesced into a single hierarchy. The Sprite naming protocol ensures that servers export their domains consistently so that all hosts, and therefore all processes, see exactly the same name space. In contrast, the V-system and Mach 3.0 use a prefix cache that is maintained on a per-process basis by library routines. While this is advertised as a feature that allows custom name spaces, I believe this is a case where generality is not what you want. Users, administrators, and developers enjoy the simplicity of a single, shared name space. The resulting, fully shared file system supports cross-compilation and easy maintenance for all architectures from any workstation.

The Sprite prefix implementation and the naming protocol are very simple. After matching a pathname against the prefix cache, the remaining pathname is sent to the server identified in the cache. The server traverses its directory structure, expanding symbolic links if necessary, until the lookup terminates or the pathname leaves its domain. A symbolic link to an absolute pathname is one way a pathanme exits, and specifying ".." in the server's root directory is the other way. If the server processes the whole pathname, it performs the requested operation (create, delete, rename, mkdir,...). Otherwise, the server returns the remaining pathname to the client. Relative pathnames bypass the prefix match and are sent to the server of the current working directory.

Mount points are handled by placing a special symbolic link at the mount point. The contents of the symbolic link is the absolute pathname of the mount point, and the link has a different file type than ordinary symbolic links so that the server knows when it hits a mount point. After expanding this link, the server returns the new pathname to the client along with an indication of how much of the pathname is the prefix of the mount point so that the client can add the prefix to its cache. Note that there is nothing in the link but its own name, so some other mechanism is used to locate the server for that name. Currently, Sprite uses

broadcasts to locate the server. After locating the server, the client reiterates the lookup procedure. Bootstrapping is achieved by broadcasting for the server of "/".

There are a number of good properties of the name space, and one limitation. First, clients are simplified. They do not iterate through directories or expand symbolic links, in contrast to the NFS naming protocol. The prefix mechanism completely replaces the UNIX mount mechanism, so servers are no more complex. The most important property is that the name space remains uniform across machines because it is the contents of the symbolic links at the mount points that defines how domains fit together, not a per-host or per-process configuration file. An advantage in common with all prefix caching schemes is that the root server is usually bypassed because clients quickly cache prefixes for the domains they use. Mount points can occur an any directory, so it is possible to nest server domains arbitrarily.

The primary limitation of the Sprite scheme is the use of broadcast to locate servers. This choice was made for simplicity, but it obviously limits the range of the name space. A general solution would be make an upcall in the case that the broadcast fails so that a user-level process can take arbitrary action to locate the server. For example, the Domain name server or some other name service could be used. This solution has the nice property that the kernel implements a lightweight mechanism (broadcast) that works for the common case, but can rely on the escape hatch to user-level in the hard case.

8 Separating Naming and I/O

A mistake that is easy to make when extending a file system to handle more than just files is to blur the distinction between the naming and I/O interfaces. The SunOS implementation, for example, has a *vfs* interface that is primarily concerned with the mount protocol, and a *vnode* interface that includes both naming and I/O operations. These interfaces stemmed from the original UNIX design where the namei procedure was the central core of name lookup. The problem is that namei handles both mount points and directory scanning. When generalizing the implementation to handle remote file systems, namei was retained and the interfaces to mounts and directory scanning were generalized. This erroneously lumps name lookup operations with I/O operations together into the vnode interface. In contrast, the Sprite prefix abstraction replaces the mount abstraction and goes further by hiding the notion of directory scanning altogether.

Consider the open system call that maps from a pathname to an I/O stream. In Sprite, this is broken into two, high-level operations: name_open and io_open. The name_open procedure returns attributes of the named object. The io_open procedure uses these attributes to create an open I/O stream. The name_open and io_open procedures may be implemented by different servers, and this is implemented cleanly by branching through the object-oriented naming and I/O interfaces. Note that a stateless protocol like NFS has no notion of an io_open operation.

The clean separation of naming and I/O allows for a number of optimizations. First, simple objects like devices can rely on a file server to implement the naming interface on their behalf. Special files are used to represent devices and pseudo-devices in the name space. Furthermore, a file server can have special files that represent devices and pseudo-devices on any machine in the network. Contrast this with NFS, which doesn't support remote device access, or even RFS, which only supports accesses to devices on the file server. These systems are limited by the vnode (or equivalent) interface that lumps naming and I/O

together.

Another obvious optimization is that in the case of regular files, name_open sets up enough state to support an I/O stream to the file. This eliminates the need for io_open to contact the file server a second time. This is a subtle, but important optimization for the common case of file access.

Both of the above optimizations are implemented cleanly by introducing a one-procedure interface on the file servers that has different implementations for different *file* types (i.e., for each kind of special file used on the server). After a file server finds a file with a general directory traversal procedure*, its name_open procedure calls through the interface to a type-specific procedure that extracts attributes from the disk-resident file descriptor and takes any special action needed for that type. It is at this point that the cache consistency protocol for regular files is invoked, for example. The attributes are returned to the client for use in calling the io_open procedure.

There are two areas in which Sprite and the Mach microkernel provide similar features, interprocess communication and user-level extensibility of a kernel-level abstraction.

9 Interprocess Communication

The Mach kernel provides a communication mechanism between threads on the same host. Threads inherit or create communication *ports* that are the destinations for messages. A *send right* to a port can be passed in a message so that communication patterns among threads can be built up. Network communication is achieved by using a user-level server, the *netmsgserver*, that maintains a mapping between local and remote ports and forwards messages over the network using a network protocol to a peer netmsgserver. Note that this design means that there are 4 user-level processes involved in a network message exchange: the two processes that wish to exchange the messages and the two netmsgserver processes that forward the message across the network. Extrapolating the microkernel philosophy, the code for the protocol stacks may also be implemented in user level. In the most general design, the function of the netmsgserver might even be separated from the implementation of the network protocol stack. In that case there would be an additional two processes involved in network communication. Another paper in this Symposium describes optimizations to this scheme [Barrera91].

In contrast, Sprite provides two special-purpose communication mechanisms, both of which are basically hidden behind the file system interface. The first is its network RPC protocol that is used solely for communication among Sprite kernels. If a kernel operation needs to be carried out on a remote machine, the kernel-to-kernel RPC protocol is used to invoke it. The other mechanism is an upcall facility that is used in a similar way, except that it forwards the operation up to a user-level process that implements a pseudo-device. The upcall mechanism is somewhat analogous to a Mach kernel using a port to communicate with a user-level external pager process. The equivalent of network RPC is not defined by the Mach kernel. Instead, it is left up to the netmsgserver implementation. Note that the two Sprite mechanisms compose nicely. If a remote, user-level process needs to be invoked, then the network RPC protocol is used first. At the remote host, the operation is converted

*. We are ignoring the effects of the prefix mechanism. This discussion applies to the final server involved in a pathname resolution, the one in whose domain the pathname terminates.

into an upcall by calling through the object-oriented interface

The network RPC protocol is based on the Birrell-Nelson RPC protocol that uses implicit acknowledgments so that ordinarily an RPC requires only two network packets[Birrell84]. Their basic model was extended to optimize bulk data transfer. Large messages are fragmented into multiple packets, and the whole batch is acknowledged by the reply packet (or subsequent request if it was the reply that was fragmented). A custom implementation allows other optimizations. An RPC request or reply is composed of two buffers plus the header. One buffer, the parameter block, is used to marshall small arguments. The other buffer refers to a large, uninterpreted block of data, usually in user-space, that can remain in place until copied onto the network by the network interface. Packet headers and parameter blocks are automatically byte-swapped at a low level, but only if the receiver has a different byte order. Packet headers contain a boot time-stamp so that crashes and reboots can be easily detected. These optimizations tune the RPC protocol for its primary use as the file system's network transport protocol.

While the RPC protocol is designed to optimize network traffic, the upcall mechanism is designed to reduce context switching and data copying. The buffer space for the upcall messages is kept in the server process's address space, not in the kernel. This allows the kernel to copy data directly between address spaces of the client application and the server process. A write on a pseudo-device can be made asynchronous at the server's option. In this case, write messages are allowed to accumulate in the server's buffer until another type of operation occurs, or until the buffer fills up. The server can also use a read buffer to decouple client reads from the generation of the data by the server. In this case, the server adds data to the read buffer as it is generated, and the kernel copies data out of the buffer in response to read operations by other processes. For example, the X server diverts mouse and keyboard input to read buffers associated with different windows, and applications read the data at their leisure. The select call is also optimized by keeping state bits inside the kernel. The kernel can test the state of a pseudo-device without a context switch to the process. The server updates the state bits as part of the upcall protocol, and it can notify the kernel directly when a pseudo-device changes state.

The Sprite RPC protocol is relatively efficient. A null call takes 2.45 msec between Sun3 class hosts, and about 1 msec between Sun4 and DECstation 3100 class hosts. This is about the same time it takes to exchange a byte of data between user-level processes on the same host using UNIX pipes. Using 16 Kbyte block sizes, Sun3 workstations can transfer data at 800 Kbytes/sec on a 10 Mbit/sec ethernet, while Sun4 workstations can achieve 900 Kbytes/sec. Other systems have implemented faster protocols. Amoeba claims the fastest RPC time in with a 1.4 msec null RPC on a Sun3 [Renesse88]. The x-kernel group reports a Sprite RPC implementation that makes a null RPC between Sun3 hosts in 1.73 msec [Hutchinson89]. The improvements by the x-kernel result from careful design of the protocol stacks, while I suspect that much of the performance of the Amoeba system stems from an assembly language implementation. The Sprite upcall mechanism has not been tuned at all, so it is about as expensive as exchanging data with UNIX pipes, or about 1 msec on a DS3100. This is the time to make a null ioctl on a pseudo-device.

10 User-Level Extensibility

The purpose of the Sprite upcall mechanism is to allow user processes to extend the kernel's

pathname and I/O interfaces. A user process can implement any semantics it chooses for a pseudo-device or a pseudo-file-system. Unlike simple message passing, the value of this approach is that general purpose features provided by the kernel, in particular network transparency, are inherited by the user-level server processes. For example, the X server lives under the pathname /hosts/*hostname*/X0. (The /hosts/*hostname* directories are just ordinary directories that are used for the few files needed on a per-host basis.) A process wishing to display a window on a particular host merely needs to open the corresponding pseudo-device. The kernel's RPC protocol is used to forward operations to the particular host. Similarly, NFS access is provided by a user-level pseudo-file-system that maps Sprite file system requests onto the NFS protocol. The server process runs on a single workstation, yet the NFS pseudo-file-system is transparently integrated into the distributed name space using the prefix table mechanism described earlier. Another feature that is inherited is blocking I/O. The server process can respond just like a device driver in order to cause the client process to block. As a result, the select system call can be used to wait on a set of devices and pseudo-devices that are located throughout the network.

Mach is similar in that the interface to a memory object can be exported to user-level [Young90]. In this case a user-level process responds to kernel requests to create and destroy memory objects, and to fetch and store pages. This facility allows a number of interesting applications, including network shared memory, compressed paging, and even remote file systems. In this case, features of the kernel-resident VM implementation are inherited by the external pager. This includes the general notion of memory objects, and more detailed features like copy-on-write.

Overall, the notion of an escape hatch to a user-level implementation is quite useful. A key difference between message passing among user processes and an upcall from the kernel is that with upcalls the kernel performs some processing on behalf of the user-level applications. The alternative in a pure message passing kernel is to put some amount of system software into runtime libraries. The fundamental difference, however, is that it is more difficult to share data structures among libraries, while the kernel has its own address space in which to maintain critical, shared data structures. Recall the differences between the kernel-resident prefix tables in Sprite vs. the per-process prefix tables in V and Mach.

Exporting a kernel interface via upcall is so useful that I have regretted the cases where it is not done. Notably, the cache consistency protocol is not exported via upcall, so caching data from pseudo-devices and pseudo-file-systems is not supported. As a result, the Sprite-to-NFS gateway provides absolutely no caching, and the Andrew benchmark runs twice as slow through the gateway as it does with a native Sprite file server. (In this case, the user-level UDP/IP server is also in the loop.) Also, as mentioned earlier, an upcall would be very useful in the case where the broadcast for a prefix fails. There is no fundamental reason why these features could not be implemented, it is just a small matter of programming.

The main drawback with Sprite's use of upcall is that ioctl is the only way to get at arbitrary functionality in the server process. Ioctl is perfectly general because it takes a command ID, and input buffer, and a reply buffer. While this is obviously clumsy, it has proved sufficient for user-level implementations of sockets and the TCP/IP protocols, terminal emulation, and the X display server. The ioctl model also assumes a request-reply pattern of interaction, while Mach ports can provide more general patterns of communication.

Configuration	Copy	Compile	Total	Penalty
DS3100 Sprite Local	22	98	120	
DS3100 Sprite Diskless	34	93	127	6%
DS3100 Mach 2.5 Local	29	107	136	
DS3100 Mach 2.5 NFS	58	147	205	50%
Sun4 Mach 2.5 Local	37	122	159	
Sun4 Sprite Local	44	128	172	
Sun4 Sprite Diskless	56	128	184	7%
Sun4 SunOS 4.0.3 Local	54	133	187	
Sun4 SunOS 4.0.3 NFS	92	213	305	63%
Sun3 Sprite Local	52	375	427	
Sun3 Sprite Diskless	75	364	439	3%

Table 1. Comparison of local and remote file system performance. The times are in seconds. The last column gives the percent slowdown of the benchmark when using a remote file system under the same OS

11 Sprite Performance.

Performance of Andrew file system benchmark, a benchmark that copies, stats, and compiles a large program, shows how well Sprite performs in the remote case. The numbers given in Table 1 were measured by Ousterhout in a series of measurements of UNIX systems [Ousterhout90]. This is a part of Table 7 from that paper.

The microkernel was unavailable to him at the time Ousterhout made his measurements. While a recent paper has reported that the microkernel with the single-server UNIX emulator can perform about as well as the Mach 2.5 kernel on the Andrew benchmark [Golub90], this makes no statement about the performance in the remote case, nor for performance with a multi-server configuration.

The primary reason for the performance advantage of the Sprite workstations is the differences in the file caching protocol. Sprite uses a delayed write strategy on both diskless clients and servers, while NFS writes data from a client through to the servers disk. The effectiveness of the Sprite caching system is presented in [Nelson88] and [Welch91]. As much as 50% of the data generated by Sprite clients is deleted before being written back to the server. This result is from long term (i.e., months long) measurements of clients that use the standard 30-second delay policy inherited from UNIX.

Recent work by Rosenblum has dramatically improved the performance of Sprite in the local case, as well. The log-structured file system aggressively optimizes writing performance, which is becoming the bottleneck as large main memory caches reduce the percentage of reads that go through to the disk [Rosenblum90][Rosenblum91].

12 The Cost of Complexity

The danger, of course, with providing a fancy distributed file system is in complexity. Consider the following sources of complexity.The cache consistency protocol relies on state maintained by the server, and this state has to be recovered after a server reboots. Users can abort operations with down servers, or they can wait for automatic recovery. During process migration the server's state has to be updated, and the semantics of shared UNIX I/O

streams (e.g., the shared seek offset) have to be maintained so that migration is transparent to the processes involved [Douglis90].Finally, the system supports a variety of "file system" objects, including devices, files, pipes, and user-level server processes.

The main cost of this complexity is in development time. As described earlier, it was about 5 years before Sprite was stable enough to be used by outsiders, although I began using Sprite for all my day-to-day work 2 years before that. Complexity doesn't necessarily imply larger programs. They do get larger as "small" features are added incrementally over time. However, major re-writes often reduce code size and simplify things. The file system benefited considerably from a re-write after initial experiences with process migration, crash recovery, and the upcall mechanism. Another of Butler Lampson's quotes is:

> "Plan to throw one away; you will anyhow." [Lampson83]

13 The Size of the Sprite Kernel

The sizes of the Sprite kernel modules are given in Appendix 1. Overall, the kernel contains about 95,000 lines of code, excluding comments, and it compiles into about 1 Megabyte on a DS3100. The largest modules are the file system (38% of lines of code), process manager (10%), which includes process migration, network and device drivers (12%), virtual memory (%8), and the RPC protocol (4%). The remaining third of the kernel is split among a miscellaneous group of modules that implement signals, synchronization primitives, the scheduler, a timer, a host monitor that triggers recovery, a debug nub to support remote debugging, malloc, free, printf, and support for profiling with the UNIX gprof program.

The file system is broken into a number of modules. The largest, **fs** (8%), contains an emulation library for 4.3 BSD system calls that used to be linked into applications via the C library. The parts of file system that directly manage disks is relatively small. **fsdm** (1%) has generic code to handle file descriptors and superblocks, while **lfs** (6%) and **ofs** (3%) implement particular block layouts. One layer higher, **fscache** (3%) maintains a cache of local and remote file blocks, while **fsconsist** (1%) implements the network consistency protocol. The **fslcl** (2%) module implements a directory hierarchy on a local disk, while **fsprefix** (1%) implements a transparently distributed name space. Local and remote implementations of various file system objects are implemented in **fsio** (3.5%) and **fsrmt** (3.5%), respectively. The **fspdev** (3%) module implements an upcall l facility. The **fsutil** (1%) module maintains the table of object descriptors (similar to vnodes) and contains other supporting routines.

Note that a Sprite kernel leaves out some things that are found in other UNIX kernels, notably protocol stacks and terminal emulation. The only network protocol in the Sprite kernel is the kernel-to-kernel RPC protocol. The TCP/IP protocols are implemented in a user-level process as a pseudo-device. The socket interface is implemented in a library that makes ioctl calls on the TCP/IP pseudo-device to setup and destroy network connections. The kernel-resident terminal driver is very crude, relying on a more sophisticated terminal emulator that runs as part of a window system. Thus Sprite takes a hybrid approach, putting performance critical features into the kernel, yet exporting the file system interface via upcall in order to allow for user-level extensibility.

14 The Size of the Mach Microkernel

Table 3 gives the sizes of the code in the directories that make up a microkernel for the DECstation 3100. For comparison, the compiled size of the emulator library and the single server are also given.The Mach microkernel is about 65% the size of a Sprite kernel in the number of non-comment code lines. This is approximately equal to leaving out the Sprite file system, although other modules in the Sprite kernel depend on the file system. The single process UNIX server is about the same size as the microkernel, although I suspect their may be some dead code in the UNIX server. I do not have access to a multi-server implementation, so I cannot comment on the size of that system. All-in-all, the code size comparison comes out a wash. By the time a file system is added back onto the microkernel, both systems are about the same size.

15 Other Comparisons

Ease of Development - A microkernel is hailed as providing an easier environment in which to develop system services. After all, they are ordinary user processes so they can be debugged in the normal ways. However, Sprite has always had a symbolic debugging facility for its kernel. The debugnub implements enough functionality to support the ptrace interface used by UNIX debuggers. It communicates with the debugger over a serial line, or over the ethernet. It is possible to set breakpoints and even single step the kernel. Debugging a multi-server environment might even be more difficult because system services are distributed into different address spaces. In this case, it isn't as easy to trace the execution of something like exec that involves the process manager, the file system, and the virtual memory system.

Virtual Memory- Mach provides an excellent internal interface to the machine-dependent MMU facilities. One of the most difficult aspects of porting an operating system is dealing with a new MMU. Much of Mach's success is due to the pmap interface and its support for multiprocessors. Sprite, too, has a decent internal interface between the machine-dependent and machine-independent parts of the kernel. It also runs on multiprocessors. However, the Mach VM interface reflects a long history of experience with VM in the Rig and Accent kernels.

Directory	Description	Procedures	Lines	Text	Bytes
boot_ufs	Bootstrap file system	30	3006		
chips	Common device code.	235	6411		
ddb	Kernel debugger	111	2836		
device	Device interface	91	3912		
inline	Compiler support	12	360		
ipc	Messages and ports	224	13097		
kern	Tasks and threads	387	11573		
scsi	Generic SCSI	113	4225		
vm	Virtual memory	137	9209		
pmax	DS3100-specific code	190	6016		
Total	A DS3100 Microkernel	1550	60645	473576	574880
Emulator	The emulator library/process	81	2421	65536	73728
Server	The monolithic UNIX server	1424	63816	503808	615600

Table 3. Sizes for a DECstation 3100 microkernel. The kernel modules are not linked separately, so I have not given compiled sizes for each module. The total size of the emulator and server are given for comparison

Real Time - The key to real time performance is a preemptible kernel, not necessarily the use of a microkernel. Early versions of Sprite had a scheduler that preempted kernel threads, although this feature was eliminated in order to simplify the scheduler. The Sprite kernel is multi-threaded internally, and with some effort the scheduler could revert to a preemptive one. Another important issue for real time is guaranteed I/O bandwidth. Again, a microkernel *per se* makes no guarantees in this regard. I suspect that the next version of SunOS will be readily converted to a real time kernel because it is multi-threaded internally and has a preemptive scheduler[Powell91]

Multiple Personalities - Mach has done a good job of extracting UNIX emulation in to a library and an associated server process. However, Sprite also emulates the BSD and Ultrix system call interfaces with a simple library. The bulk of UNIX compatibility is in the file system, which is already provided. Another approach to emulation is taken by the Chorus microkernel. Different system services are downloaded into the kernel depending on the needs of the applications [Guillemont91]. I prefer this approach to the Mach multi-server approach because I feel that kernel-resident modules will perform better. As for Mac and DOS emulation, I do not see a real need for this given the multi-tasking facilities available with Windows 3.0 and Mac's System 7. Only a small set of programmers will want to program UNIX-style on DOS or a Mac, while commercial developers and users will almost certainly prefer their native DOS and Mac environments.

16 What I Would Do Differently

If I were to write a new operating system today (which I am not), I would like to pick and choose from the excellent work done by various research groups. For example, the Mach pmap interface, the x-kernel protocol stacks, the Sprite distributed file system, and Bershad's LRPC would be an excellent starting point. The other crucial factor is support from the hardware vendors in the form of device drivers and bootstrap code. There are a number of difficulties with this approach, however. A practical matter is that each of these have been developed in their own environment, so integration would be difficult, even if it were possible to get the source code. The other, central problem is that the interfaces to these subsystems are not always well defined. Often the top-level interface is published, but the interfaces relied on by the implementations are not specified.

The next time around I would still not choose message passing. Instead, I would provide two communication mechanisms in the kernel, LRPC for local communication and network RPC for remote communication. I believe that the procedure call model is right abstraction to present to programmers. I would provide, in the kernel, a high-level distributed name space using a prefix table mechanism. I would retain the file system's I/O interface as the default interface, but I would also allow other interfaces to be associated with pathnames. Thus the kernel would still provide the default, high-performance file system. However, user processes could use the hierarchical name space for binding to arbitrary interfaces implemented by other, user-level processes. Finally, I would leverage more on dynamic loading so that only the needed kernel modules need to be loaded on any particular machine, and so that turn-around time in development would improve.

17 Conclusions

First, while it is always true that you can build things with general purpose, low-level facilities, it is also true that choosing a higher-level abstraction gives you more flexibility in the

implementation. A higher-level abstraction also does more for its clients, although it may restrict them in some ways. For Sprite we chose the file system interface as opposed to a message passing interface. As a result, we had the flexibility to implement simple, efficient mechanisms for its transparently distributed name space and its remote file access. The file system also provides an overall structure to the system that provides a benefit in uniformity and simplicity. The danger in choosing a high-level abstraction is that it may not be appropriate for every application. However, in the case of the file system I can safely assert that this abstraction is useful to a wide range of applications.

Second, while it is important for performance to provide kernel-level support, an upcall mechanism that integrates user-level processes into kernel abstractions is a good idea.The kernel can provide the basic framework for the system, such as a high-level name space or a copy-on-write VM facility, yet the user-level processes can extend the system in new ways without modifying the kernel. This is different than pure message passing where the kernel makes no statements about the overall structure of the system. By now, I think we know enough to provide some higher-level functionality in the kernel.

18 Acknowledgments

I would like to that all the helpful feedback from the reviewers, especially Dave Nichols. I would like to thank the program committee and Alan Langerman for giving me the opportunity to present these ideas to this audience.

19 References

Accetta86. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young, Mach: A New Kernel Foundation for UNIX Development. *Proc. of the Summer 1986 USENIX Conference*, July 1986.

Baker91. M. Baker, J. Hartman, M. Kupfer, K. Shirriff and J. Ousterhout, Measurements of a Distributed File System, *(to appear) Proc. of the 13th Symp. on Operating System Prin., Operating Systems Review*, Oct. 1991.

Bershad91. B. N. Bershad, T. E. Anderson, E. D. Lazowska and H. M. Levy, User-Level Interprocess Communication for Shared Memory Multiprocessors, *ACM Transactions on Computer Systems*, 9, 2 (May 1991), 175-198.

Birrell84. A. D. Birrell and B. J. Nelson, Implementing Remote Procedure Calls, *ACM Transactions on Computer Systems* 2, 1 (Feb. 1984), 39-59.

Clark85. D. Clark, The Structuring of Systems Using Upcalls, *Proc. of the 10th Symp. on Operating System Prin., Operating Systems Review* 19, 5 (Dec. 1985), 171-180.

Douglis90. F. Douglis, Transparent Process Migration for Personal Workstations, PhD Thesis, Sep. 1990. University of California, Berkeley.

Draves90. R. Draves, A Revised IPC Interface, *Proc. of the Mach Workshop*, Oct. 1990, 101-121

Feirtag71. R. Feirtag, E. Organick, The Multics Input/Output System, *Proc. of the 3rd SOSP*, 1971, 35-41.

Golub90. D. Golub, R. Dean, Forin and R. Rashid, UNIX as an Application Program, *Proc. of the Summer 1990 USENIX Conference*, June 1990, 87-96.

Guillemont91. M. Guillemont, J. Lipkis, D. Orr and M. Rozier, A Second-Generation Micro-Kernel Based UNIX; Lessons in Performance and Compatibility, *Proc. of the Winter 1991 USENIX Conference*, Jan. 1991, 13-22.

Hill86. M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A.

Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, Design Decisions in SPUR, *IEEE Computer* 19, 11 (November 1986).

Hutchinson89. N. C. Hutchinson, L. L. Peterson, M. B. Abbott and S. O'Malley, RPC in the x-Kernel: Evaluating New Design Techniques, *Proc. 12th Symp. on Operating System Prin., Operating Systems Review* 23, 5 (December 1989), 91-101.

Barrara91. J. S. Barrera. III, A Fast Mach Network IPC Implementation, *(to appear) Proc. of the 2nd Mach Symposium*, Nov. 1991.

Lampson83. B. Lampson, Hints for Computer System Design, *Proc. 9th Symp. on Operating System Prin., Operating Systems Review* 17, 5 (October 1983), 33-48.

Lauer81. H. C. Lauer, Observations on the Development of an Operating System, *Proc. of the 8th Symp. on Operating System Prin., Operating Systems Review* 15, 5 (Dec. 1981), 30-36.

Li89 .K. Li and P. Hudak, Memory Coherence in Shared Virtual Memory Systems, *ACM Transactions on Computer Systems* 7, 4 (Nov. 1989), 321-359.

Nelson88. M. Nelson, B. Welch and J. Ousterhout, Caching in the Sprite Network File System, *ACM Transactions Computer Systems* 6, 1 (Feb. 1988), 134-154.

Ousterhout90. J. Ousterhout, Why Aren't Operating Systems Getting Faster As Fast as Hardware?, *Proc. of the Summer 1990 USENIX Conference.*, June 1990, 247-256.

Powell91. M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein and M. Weeks, SunOS Multi-thread Architecture, *Proc. of the Winter 1991 USENIX Conference*, Jan. 1991, 65-80.

Renesse88. R. Renesse, H. Staveren and A. W. Tanenbaum, Performance of the World's Fastest Distributed Operating System, *Operating Systems Review* 22, 4 (Oct. 1988), 25- 34.

Rosenblum90. M. Rosenblum and J. K. Ousterhout, The LFS Storage Manager, *Proc. of the Summer 1990 USENIX Conference*, June 1990, 247-256.

Rosenblum91. M. Rosenblum and J. K. Ousterhout, The Design and Implementation of a Log-Structured File System, *(to appear) Proc. of the 13th Symp. on Operating System Prin.*, Operating Systems Review, Oct. 1991.

Welch86a. B. B. Welch, The Sprite Remote Procedure Call System, Technical Report UCB/Computer Science Dpt. 86/302, University of California, Berkeley, June 1986.

Welch86b. B. B. Welch and J. K. Ousterhout, Prefix Tables: A Simple Mechanism for Locating Files in a Distributed Filesystem, *Proc. of the 6th ICDCS*, May 1986, 184-189.

Welch88. B. B. Welch and J. K. Ousterhout, Pseudo-Devices: User-Level Extensions to the Sprite File System, *Proc. of the 1988 Summer USENIX Conf.*, June 1988, 184-189.

Welch90. B. B. Welch, Naming, State Management, and User-Level Extensions in the Sprite Distributed File System, PhD Thesis, 1990. University of California, Berkeley.

Welch91 .B. B. Welch, Measured Performance of Caching in the Sprite Network File System, *(to appear late '91) Computing Systems.*

Young87. M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. CHew, W. Bolosky, D. Black and R. Baron, The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System, *Proc. 11th Symp. on Operating System Prin., Operating Systems Review* 21, 5 (Nov. 1987), 63-76.

Module	Description	Procedures	Lines	Text	Bytes
dbg.ds3100	Debug Nub	27	1509	13680	24672
dev	Devices	117	4159	83296	127472
dev.ds3100	DS3100 drivers	67	5350		
fs	FS syscalls & BSD compat.	158	8018	76784	97296
fscache	Cache manager	70	2747	24032	27184
fsconsist	Consistency protocol	35	1338	11584	14080
fsdm	Local disk manager	8	889	5904	7344
fsio	Local I/O objects	100	3490	26352	31760
fslcl	Local directories	43	2086	16896	19104
fspdev	Upcall to user-level	68	3155	24784	28176
fsprefix	Distributed naming	29	1200	10704	11824
fsrmt	Remote I/O objects	91	3644	28816	32336
fsutil	Table management	62	1547	13792	16464
lfs	Log Structured FS	132	5394	46096	52064
libc	Printf, etc.	80	5133	39104	57056
libc.ds3100	ditto	1	99		
mach.ds3100	Trap handlers & Ultirx compat.	135	5595	63840	80384
main.ds3100	System start-up	3	294	2480	3648
mem	Malloc/free	17	841	6128	22544
net	Network interface	42	1718	37760	45408
net.ds3100	Network devices	18	875		
ofs	Old disk layout	51	2724	23360	2744
proc	Process manager	225	9747	77776	96000
proc.ds3100	ditto	1	52		
prof	Kernel profiling	8	198	2464	3904
prof.ds3100	mtrace, etc.	7	288		
raid	Disk arrays	116	3230	32 *	128 *
recov	Network recovery	40	1253	10448	12144
rpc	Remote Procedure Call	79	3709	38704	50928
rpc.ds3100	ditto	1	8		
sched	Scheduler	24	814	9440	11888
sig	Signals	31	1189	9200	1097
sync	Monitors, condition vars	42	1493	16152	17348
sys	Syscalls and miscellany	38	1847	9312	17184
timer	Timer and callout	17	463	8560	15168
timer.ds3100	Timer device	11	247		
utils	Hash and Trace	17	710	6656	9200
utils.ds3100	ditto	3	39		
vm	Virtual Memory	198	6565	77328	103200
vm.ds3100	ditto	49	1068		
TOTAL	A DS3100 Sprite kernel	2261	94725	828576	1085632

Appendix 1. Module sizes for the Sprite kernel. The count of lines of code exclude comment blocks and cpp directives. The Text and Bytes columns give the text size and total size of the compiled modules as reported by the UNIX size command. The compiled sizes are for the DECstation 3100. Machine-dependent modules are named with a .ds3100 suffix. They are linked together with corresponding machine-independent modules so separate compiled sizes are not given. Compiled sizes for the Sun3 are roughly 75% the size for the other. RISC-based architectures. The presence of compatibility code in both fs and mach.ds3100 is because the compatibility code is being rewritten. There is considerable overlap between the two.

* raid is only compiled for the Sun4 architecture. The compiled sizes are for stubs.

Overview of the Architecture of Distributed Trusted Mach

E. John Sebes
Trusted Information Systems
444 Castro Street, Suite 800
Mountain View, CA 94041
ejs@ba.tis.com

Abstract

The Distributed Trusted Mach (DTMach) Concept Exploration produced a design that extends Trusted Mach (TMach), which is TIS's development of a B3 trusted version of the Mach operating system. The overall goal of DTMach is to integrate distributed systems functionality with the security mechanisms of TMach. TMach's modifications to the interprocess communications mechanisms are combined with transparent network communications based on similar functionality from Mach. In addition, DTMach defines extensions to the TMach system servers that enable them to utilize this distributed IPC to provide their services in a distributed manner.

Introduction

Distributed Trusted Mach (DTMach)[1] incorporates the previous work of Mach and Trusted Mach (TMach)[3][4][5], and extends them to provide the basis for a trusted distributed operating system designed to meet the B3 security requirements[6]. This paper provides a summary of the DTMach architecture and key design features.

The term "distributed system" has many definitions, but we take the working definition of a system in which: the presence of separate nodes in the system is largely invisible; system operations have the same functional behavior regardless of which node they are invoked from; and system resources are managed on a system-wide basis.

The Mach system was designed to support distribution, in that system resources can be utilized from any of several Mach nodes connected by a network. However, Mach is not a distributed system in the above sense. This is due to the fact that when system resources are not local to a user's node, the user must frequently take this fact into account when accessing those resources. In other words, Mach system software does not yet take full advantage of Mach features to make system service fully transparent and uniform across nodes in a network.

TMach, on the other hand, is a *stand-alone* trusted system that is derived from and compatible with Mach. TMach does provide for the use of network devices, and allows for untrusted implementations of network protocols. However, it is beyond the current scope of the TMach development to support the trusted communication between TMach nodes that would support the extension of TMach system services throughout a networked system. Therefore, TMach does not address trust enhancement for the Mach functionality that supports distribution.

The purpose of DTMach, therefore, is two-fold. First, DTMach must extend TMach to provide trust enhancements for the same sort of distribution support that Mach provides. Secondly, DTMach must build on these extensions to actually provide, in a trustworthy manner, the distribution of services that Mach itself does not provide.

To achieve the purposes described above, DTMach combines TMach trusted IPC with Mach's distributed IPC, and adds extensions to ensure the security of the combined trusted distributed IPC. In keeping with the multi-server architecture, this functionality is provided not by kernel modifications, but by a server called the Network Server. This service provides the basis for trusted networked virtual memory management, which, together with the extended IPC, forms the foundation on which trusted distributed system servers can be built. DTMach defines a general approach to a distributed system service, designed to allow for the extension of TMach system servers without requiring extensive redesign. Using this approach, each of the TMach system servers will be extended to allow its service to be provided in a distributed manner.

DTMach Architecture

The software components of DTMach fall into three main layers:

- The TMach kernel;
- The new DTMach components: the Network Server and the Network Pager;
- The TMach system servers, extended to provide distributed service.

The TMach kernel is used in DTMach, and needs no additional functionality to handle the extension of kernel services into a distributed environment. Instead the Network Server and Network Pager tasks provide the distribution of kernel services — IPC and memory management, respectively. The only change required to the TMach kernel is an additional privilege needed by the Network Server, described below. This privilege can be provided by the existing TMach kernel privilege mechanism.

The Network Server is the DTMach component that provides the functionality of trusted distributed inter-process communication (IPC). The TMach kernel provides trusted IPC, but only on a single machine. In Mach, the kernel's IPC is extended in a distributed manner by the Mach NetMessage Server, but this IPC lacks the security features of TMach. The DTMach Network Server combines and builds on both of these areas of previous work. As a result, DTMach IPC is the same as TMach IPC, but is distributed in a manner similar to that of Mach[7][8].

Just as the Network Server extends TMach IPC in a distributed manner, the Network Pager extends TMach virtual memory management so that the same services are available in a distributed environment. TMach memory management is built on the same basic functionality of ports and messages that constitute IPC. As a result, the Network Pager uses the trusted distributed IPC provided by the Network Server.

Both the Network Server and Network Pager extend kernel functionality in a way that is invisible to the users of kernel services. For example, when sending an IPC message, a client task cannot determine whether its final destination is on the same node– with the IPC service provided by the kernel– or on another node, with the service extended by the actions of the Network Server. In general, the involvement of the Network Server and Network Pager in providing these "kernel-level" services is invisible to clients. As a

result, we say that the Network Server and Network Pager operate at the "kernel layer" of the system. That is, they provide the same services as the kernel, but in a distributed manner. However, they can do so without having to execute in the same privileged state and hardware-protected address space in which the kernel executes.

While the TMach kernel provides the building blocks of operating system services, the bulk of operating system services are provided by the TMach system servers. The same is true of DTMach. However, in DTMach, the TMach system servers are extended so that each server on each node not only provides its service on that node, but also co-operates with the same server on other nodes, in order to provide the service in a distributed manner. For example, the TMach File Server provides services for the use of files that are physically located on the node that the File Server runs on. In DTMach, the File Server also does so, but also communicates with other File Servers on other nodes. As a result, a client can contact the File Server on its node, and request service for any file without regard for which node the file is physically located on; if the file is on a remote node, the local File Server can arrange for the service request to be forwarded to the appropriate remote File Server. The basis of this cooperation and communication is the trusted distributed IPC provided by the Network Server, and distributed VM provided by the Network Pager.

Network Server

The primary function of the Network Server is to act as intermediary and message transport facility in IPC transactions where the sender and receiver are on different nodes in a DTMach system. The overall approach to message handling is similar to that in Mach [7][8], as is the approach to communication between the various instances of the Network Server on the various nodes in a system. However, the IPC mechanism supported is that of TMach, and an essential requirement of the DTMach Network Server is to enforce relevant aspects of the TMach security policy.

Network Server Architecture

The DTMach Network Server is composed of three separate parts, each of which is implemented in a separate task:

Net Message Server: is similar in overall functionality to the Mach Net Message Server, but has several significant additional security-related requirements.

Net Protocol Server: implements the various network communication protocols used by the Net Message Server.

Net Line Server: manages the network devices.

The essential reason for this breakdown is that it allows the Net Protocol Server to be an untrusted component. That is, the trusted Net Message and Net Line Servers do not rely on the secure operation of the Net Protocol Server. Therefore, the Net Protocol Server can not violate any aspects of the TMach security policy. The Net Message Server does have security-related requirements, including the assurance that security-relevant data about each message (e.g. the label and identity of the sender) is accurately sent with each message. The Net Line Server also has security-related requirements; the network devices are multi-level devices, and must be managed by a trusted component to ensure that the devices are used in accordance with the TMach security policy.

This arrangement is made possible by certain protection measures enforced by the Net Message and Net Line Servers on the Net Protocol Server. Details of this approach are given in [2]. As a result, existing network communication protocols and protocol implementations can be used in DTMach, without any re-engineering for trust-related functionality or assurance requirements.

The interaction of these three components in one IPC transaction is summarized as follows:

1. The Net Message Server receives a message destined for another node.
2. The Net Message Server determines where and how to send the message, and passes it to the Net Protocol Server.
3. The Net Protocol Server breaks the message up into packets suitable to send over a network, and sends each to the Net Line Server.
4. The Net Line Server applies integrity measures to each packet, and writes it on the network device.
5. The packets arrive on the destination node.
6. The Net Line Server on the destination node reads packets from the network device, checks the integrity of each packet, and passes each intact packet to the Net Protocol Server.
7. The Net Protocol Server on the destination node re-assembles packets into a message, which it passes to the Net Message Server.
8. The Net Message Server on the destination node delivers the message to its destination task.

Network Server Interactions

Each DTMach node has an instance of the Network Server, which has three main kinds of interactions: interactions with clients, with the local kernel, and with remote Network Server instances.

The interactions with clients are based on the port management strategy of the Network Server: on each node, the Network Server instance is the receiver for local ports for which the actual receivers are on other nodes. In other words, the Network Server is the local stand-in for remote tasks to which local tasks can send IPC messages. As a result, the Network Server can both: receive from local clients every message bound for another node; and, send to the proper local clients each message received via the network from a Network Server instance on another node.

The intra-Network-Server interactions center around the transport of IPC messages from one node to another. In addition to exchanging such messages, Network Server instances also exchange control messages that track changes in system-wide IPC data, such as the senders and receivers of networked ports.

Figure 1 shows these two interactions. Task A sends over a port a message intended for another client task. However, that client is on another node. Therefore, the receiver of the port is, unknown to Task A, the local Network Server instance, which sends the message to the Network Server instance on the proper remote node. There, the Network Server sends

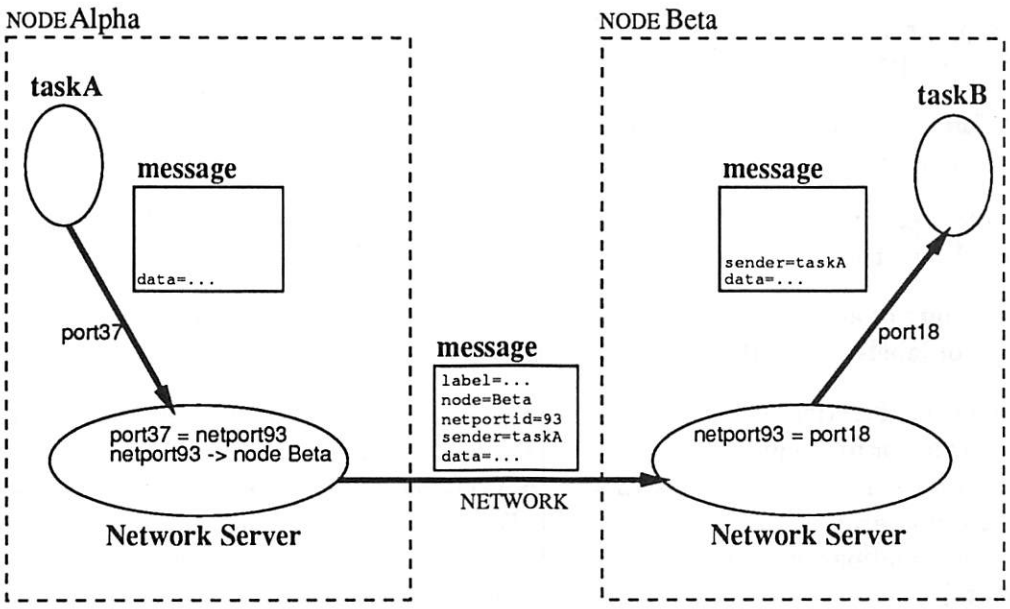


Figure 1: DTMach Message Interactions

the message over a local port to the final receiver of the message. Furthermore, the message send is done in such a way that it appears that the sender was Task A, rather than the Network Server.

The Network Server's interactions with the kernel arise from the kernel's role as the provider of the local IPC service. Therefore, the Network Server does not directly interact with local clients; rather, it uses the kernel's service to send and receive messages from clients. Furthermore, the Network Server requires a privilege from the kernel, which the Network Server uses when it sends a message to a local client in order to deliver it. Normally, the message as received by the client task would indicate that the sender was the Network Server task. Using this privilege, however, the Network Server can tell the kernel to set the message's sender-identity to that of the original sending client task on the remote node.

Figure 1 also shows some of the Network Server's security-related functionality. For each message it moves between nodes, the Network Server adds security-related information to the message — information that is stripped off and used by the receiving Network Server instance. These are schematically shown in the figure as the "label" field of the message in the middle of the figure. In reality, this information includes not only the security label of the message, but also information about the identity of the sender. This information is used on the receiving node to enforce the TMach security policy.

Other Network Server Functionality

In addition to the key message passing and security-related functionality outlined above, there are a variety of other functional requirements of the Network Server:

- Mutual authentication between Network Server instances on different nodes.

- Cryptographic services required for this authentication (both of these are described in more detail in [1]).
- Enforcement of message non-disclosure and integrity protections against incorrect behavior of the untrusted Net Protocol Server ([2]).
- Protection of the Net Protocol Server task, to prevent tampering by clients which could result in denial of service.
- A variety of administrative co-ordination, for example ensuring consistent interpretation of labels across the distributed system.

In all of this functionality, the Network Server assumes that the network is protected, either physically, or by cryptographic devices attached to the network. Lacking physically protected networks, DTMach systems that carry classified information must use government-approved cryptographic devices. In cases of DTMach systems handling non-classified but sensitive information (including commercial environments and non-military government environments) where the network is physically unprotected, the non-disclosure and integrity of network data can be ensured by packet-level encryption carried out by the Net Line Server. The necessary cryptographic services would already be part of the Network Server for purposes of mutual authentication.

Network Pager

The Network Server's IPC service plays a key role in DTMach's extension of the TMach kernel's memory management services into a distributed environment.

As in Mach, memory management in TMach is split between the kernel and tasks called pagers. The External Memory Management Interface (EMMI) defines how the kernel and pagers communicate in order to fill their respective roles in the management of memory objects. Memory objects are kernel-defined objects that represent a region of memory which may be mapped into one or more task's address space. The EMMI also defines security constraints which ensure that operations on memory objects cannot violate the TMach security policy. The kernel enforces these constraints, as well as constraints which prevent untrusted pagers from violating the policy.

Operations on a memory object (by pagers or clients of pagers) are made using operations on the kernel-managed port which represents the object. Since memory objects are represented by memory object ports, and the EMMI is carried out by IPC, it is a simple matter for memory management communication to be carried out over a network. Since the IPC is transparent, so is the memory management communication. Hence, the distributed IPC also serves as a basis for distributed memory management. Thus, in a distributed environment, an external pager may reside on a different node from some of its clients. In this case, the kernel's communication with the pager is just like any other component using distributed IPC. The local port that the kernel sends messages on is connected via the NetMessage Server to the port that the pager receives kernel messages from. The messages and responses are exactly the same as if the external pager were a local process.

In addition to operations on memory objects, TMach memory management also governs the use of memory references (or "out-of-line data") in IPC messages. When a message is sent on a port for which the real receiver is on a remote node, the Network Server acts as an intermediary, as described above. When such a message has a memory reference in it, then

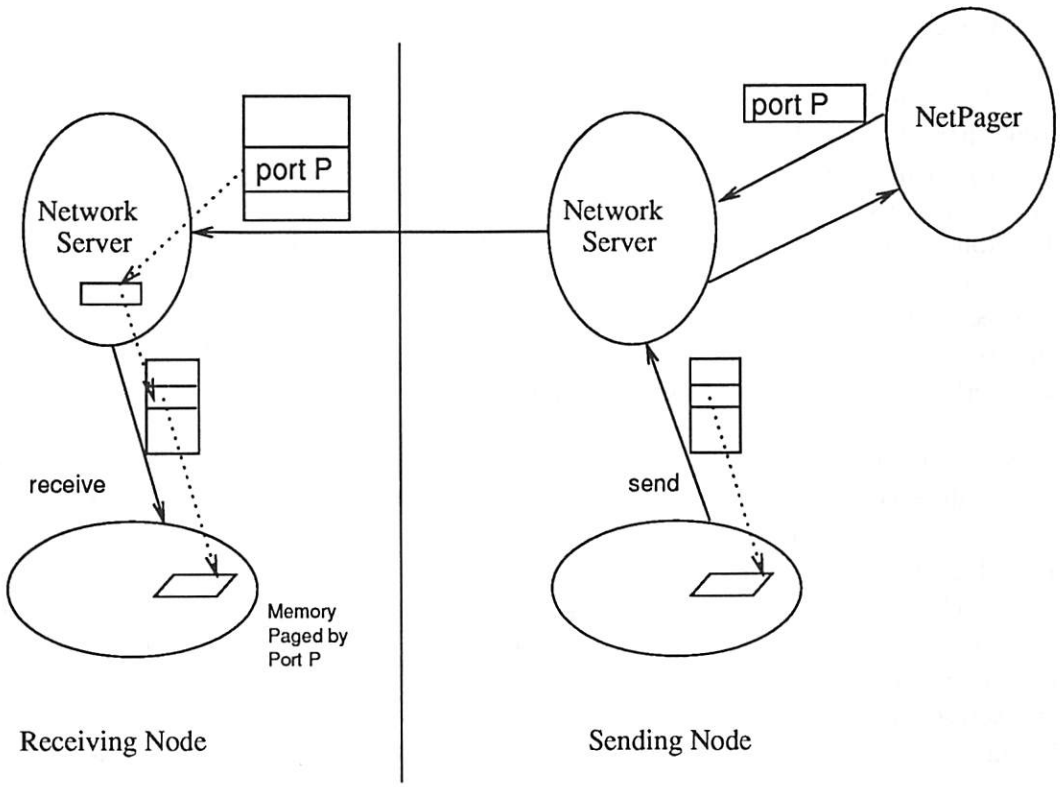


Figure 2: Network Pager Scenario

the Network Server must take action to ensure that the memory reference is meaningful on the destination node. It does so by calling on the Network Pager, the function of which is to manage such out-of-line data by acting as a pager for it.

Each node has a Network Pager task, just as each node has a task that is an instance of the Network Server. The services of a Network Pager task are initiated only by the local Network Server instance, although subsequently the Network Pager communicates with other tasks as part of its paging duties. As a result, the Network Pager can be considered as an "outboard" part of the Network Server, which is implemented as a separate task in order to minimize the duties of the Network Server proper.

The basic interactions between the Network Server and Network Pager are shown in Figure 2. In the figure, the client task in the lower right sends a message over a networked port, and the message has a memory reference in it (indicated by a box with an arrow pointing back to the client's address space). The Network Server receives the message, and notices the memory reference. The Network Server sends to the Network Pager a message containing the memory reference. The Network Pager returns to the Network Server a port representing this memory. The Network Server then takes the original client's message, and constructs from it a message in which the memory reference in replaced by the memory port. This is the message which is transported across the network to the remote Network Server, which delivers it to the destination client task as a memory reference. From then on, the remote client's use of the memory reference proceeds normally, even though the management of the referenced memory is actually carried out in part by the remote Network Pager.

The trust issues related to distributed memory management are largely related to the way the distributed memory management is used by other components, rather than to the details of the mechanisms. That is, the distributed memory management mechanisms described here do not add any new requirements for constraints or checks, other than those already a part of the existing EMMI definition, and the IPC service. For example, a client task may receive virtual memory only from a pager at the same security label, or from a trusted multi-level pager (such as the Network Pager). This rule is enforced by the kernel for local memory management operations; it applies and is enforced just the same in the distributed case as well.

One issue is dependency layering. To meet B3 requirements, the Network Pager (and the Network Server, on which it depends) must be implemented in such a way that they do not depend on any other TCB components which use their services– otherwise, there could be a circular functional dependency which could violate the required architectural layering. This issue does constrain the design of the Network Server and any Network Pagers, but the design in DTMach obviates the difficulty by having these components depend only on the kernel, and not use any of the services of other TCB components.

Covert channels are also a security issue, because pagers are a likely source of covert channel activity. Pagers gauge the paging activity of their clients and the demands on physical memory. Tasks running on the same node as a pager may be able to gauge the pager's activity and infer information about the activity of the clients.

Finally, security labeling is an issue, because the labels of the memory objects passing over a network must be kept consistent across the various nodes in the system. This is handled on two fronts. First, the EMMI definition and the Network Server functionality are sufficient to accurately move the label representation along with memory objects and memory references. Second, administrative measures are necessary to ensure that the same label representation on various nodes actually denotes the same label value on all those nodes.

Servers

In DTMach, as in TMach and Mach, most of the operating services are implemented in servers. DTMach servers build on the existing TMach servers, adding functionality that takes advantage of the distributed IPC and VM services so that servers on different nodes can communicate with one another. As a result, the communicating servers can provide a distributed service.

A distributed server is composed of a number of tasks (which we call server instances) on various nodes. For example, the distributed File Server in a very simple, small DTMach system might be composed of a File Server instance task on each node. Each of these tasks would be similar to the TMach File Server, with the addition of functionality to communicate with other instances, and to propagate shared information.

Approaches to Distributed Services

There is a range of options for expanding TMach system services in a coordinated manner among a collection of networked hosts. In the minimal case, the only distributed server would be the Network Server (which is by definition a distributed server). It would serve to connect the central node providing most services with other "server-less" nodes via

distributed IPC. This approach is undesirable because it obviously presents a large problem with the essential distributed system goal of reliability. Also, the approach is untenable at larger scales because the central node is a bottleneck for service.

Another approach is to have all the nodes have a full complement of TMach servers, of which only the various Name Servers communicate with one another. In TMach, the Name Server manages the entire system's name space of objects, and is the focal point for all requests for access to objects. In this approach, there would not be a true distributed Name Server, however. Rather, a system-wide name space would be created using the TMach feature of the "mount point". One central Name Server would be the root of the entire name space, and the other Name Servers on the other nodes would mount their local name spaces onto the central one. The Name Servers would use distributed IPC to forward to one another requests from local clients for remote objects. Other servers would communicate only with local servers. The problem with reliability, while ameliorated somewhat, is still significant. The problem with scaling and performance, while different in some specifics from the previous approach, is also significant. However, this approach is technically workable on a small scale.

Goals for Distributed Services

The deficiencies of approaches such as these lead to the need for fully distributed servers. Such cross-node intra-server cooperation is necessary to provide service that is:

- truly distributed, e.g. largely transparent with respect to the existence of different nodes;
- meets trust requirements;
- meets reasonable goals for distributed system properties.

The goals for DTMach's distributed services take into account both trust requirements, real-world usability, and the limitations imposed by the interactions of the various goals. That is, it would be desirable to build a highly reliable, high-performance, large scale system, but that would be beyond the scope of trying to combine trust with distributed system functionality. Instead, the goals are to create a system which:

- meets B3 trusted system requirements;
- can scale up to approximately 100 nodes (though not larger);
- can respond to some multiple points of failure, (though not to many simultaneous points of failure);
- can operate on interconnected LANs (though not in large internetworked environments);
- can provide very tight consistency among some system-wide databases;
- has production-quality performance.

Server Issues

For each server, the effects of these requirements and goals are somewhat different. In some cases, the effect is the same. For example, reliability is an issue common to all distributed servers, in which each instance of a trusted server will depend on other instances to cooperatively provide service. However, each instance must also be prepared to deal gracefully with system failures, in which other instances may become unavailable. In particular, each instance must continue to operate correctly and securely, independently of its connection to other instances. In some situations, of course, this may mean that the instance provides limited service, or no service at all.

In other cases, different issues will affect different servers quite differently. For example, consistency of server data is a critical issue for servers whose internal data is security-critical, such as authentication data. For these servers, handling system-wide databases must ensure the complete consistency of the data. For other trusted servers which do not have security-critical data, internal database consistency can be maintained more flexibly.

The following provides a summary of how each server design deals with some of the major points.

Name Server is a distributed server, which has an instance on all nodes. Each instance is the central point of contact for object access for all the clients on that node. Much of the co-operation between instances is oriented to maintaining a system-wide name space that is identical throughout the system.

The Name Server provides services for some items in the name space (such as directories); for items of other types (e.g. files), the management is provided by another server. However, for all items, the clients' initial access requests are approved or denied by the Name Server; each approved request is either handled by the Name Server itself (e.g. for directories), or forwarded to another server in cases where the Name Server does not provide service for the item. Additionally, each server instance might not provide service for a particular item, even if the Name Server does provide service for items of that type. In such cases, the instance forwards access requests to the instance that does manage the particular object.

Particular items that are managed by the Name Server can be replicated, i.e. simultaneously managed by more than one instance of the Name Server. For replicated items, read access can be provided locally for each of the nodes whose Name Server replicates the object. However, strict cooperation between replicating server instances is required for modification of replicated items. This cooperation must not only prevent conflicting multiple writes, but also ensure that the updated object data becomes effective at the same time in each participating instance. This is required for the Name Server because the objects that it manages contain security-critical data— such as Access Control Lists of directories– which must not become inconsistent.

Authentication Server is also a distributed server, which typically would have an instance on a relatively small portion of the nodes in a DTMach system. Multiple instances are required, so that authentication service (e.g. login) is reliably available. Since the authentication service should be the same throughout the system, the various instances must maintain a shared authentication database. This is a security-critical database — one that changes, albeit rather slowly, and only under the control of trusted administrative personnel.

Therefore, the main addition to existing TMach Authentication Server functionality is the cooperation required to maintain this database with strict consistency. This approach is required so that the security policy is enforced in the same way throughout the system. However, it has inherent problems at large scales, where it may frequently be the case that all of the Authentication Servers are not available to cooperate on database updates. Successfully dealing with this conflict of interest will require future work in defining manageably-sized administrative domains (in which the scaling problem is avoided) which can inter-operate for authentication (to provide consistent authentication service).

File Server may be a distributed server, with instances on every node that has mass storage devices used to store information useful throughout the system.

However, a distributed File Server is not a strict requirement for a distributed system, which could have separate stand-alone TMach File Servers which only know about the files resident on the local node. Distributed access to files is provided, in any case, by the Name Server, since the Name Server can forward file access requests to the Name Server instance on a node where the file is resident. This Name Server instance forwards accepted file access requests to the local File Server. That local File Server need not be aware of the fact that the request came from another node.

A distributed File Server is a real benefit, however, so that it can provide replication services, and provide more flexible routing of file access requests, relieving the Name Server of the responsibility of correctly determining the location of files.

The main trust issue of the File Server pertains to maintaining system-wide consistency of files which are used by trusted servers to store security-critical data.

Audit Server exists in TMach to provide a repository for all audit data on a TMach system. In DTMach, there is typically an Audit Server on every node in a system. In some cases it is possible for an Audit Server on one node to act as the repository for data for other nodes, but network reliability problems can make it difficult to meet trust requirements for audit. As a result, it is simpler to have an Audit Server on every node.

The Audit Server may or may not be a distributed server. If not, then each system has a separate server, and other means must be used for the required functionality of system-wide aggregation and analysis of audit data. A distributed Audit Server, however, would have an instance on each node. These instances are essentially TMach Audit Servers, with functional additions for communication in support of system-wide aggregation and analysis.

Type Server manages the items that represent the types of other (non-type) items in the name space. For system-defined types, the type data is static. Therefore, there is no need for dynamic consistency cooperation of type data. User-defined type data can be added or changed, but exact consistency is not required for security. Therefore, looser consistency mechanisms could be employed in a distributed Type Server. Since the Name Server and the Type Server work so closely together, it is easiest for the Type Server to be distributed in the same manner as the Name Server.

Other Servers Other TMach servers need not be distributed servers in DTMach. For example, the Device Server manages the allocation of control of local devices to local

tasks, and there is no need for a Device Server to know anything about devices on other nodes.

Summary

Distributed Trusted Mach extends the IPC and VM services of the TMach kernel, so that the services can be provided in a distributed environment. This extension requires no functional changes to the kernel, since the distribution functionality is provided by tasks called the Network Server and Network Pager. These services provide the basis for the distribution of the rest of the system services, which are implemented in tasks called servers. TMach system servers provide these services in a manner consistent with B3 trust requirements. In DTMach, some of these servers are extended to be distributed servers. These distributed servers have instances on several nodes, and these instances cooperate to provide services in a distributed manner. The resulting design promises a system which combines trust features and distributed system functionality, while meeting moderate distributed systems goals and stringent trust requirements.

Acknowledgments

The work described in this paper resulted from the efforts of: L. Badger, M. Bernstein, R. Feiertag, N. Kelem, W. Morrison, H. Orman, D. Rothnie, E.J. Sebes, G. Skinner.

References

[1] Trusted Information Systems, "Distributed Trusted Mach Concept Exploration Final Report," Rome Air Development Center, 1990. TIS Rep. 374.

[2] E. John Sebes and Richard J. Feiertag, "Trusted Distributed Computing: Using Untrusted Network Protocols," *Proceedings of the 14th National Computer Security Conference*, October 1991.

[3] M. Branstad, H. Tajalli, F. Mayer, "Security Issues of the Trusted Mach System," Rep. 138, Trusted Information Systems, January 1988.

[4] M. Branstad, H. Tajalli, "Security Policy for the Trusted Mach Kernel," Rep. 179, Trusted Information Systems, September 1988.

[5] M. Branstad, H. Tajalli, F. Mayer, D. Dalva, J. Graham, "Access Mediation in Trusted Mach," Rep. 203, Trusted Information Systems, March 1989.

[6] National Computer Security Center, "Trusted Computer System Evaluation Criteria," DoD 5200.28.STD, December 1985.

[7] R. D. Sansom, *et al*, "Extending a Capability Based System into a Distributed Environment," *Communication of the ACM*, February 1986.

[8] R. D. Sansom, "Building a Secure Distributed Computer System," Carnegie-Mellon University Rep. CMU-CS-88-141, 1988.

The USENIX Association

The USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environments, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital.

To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics; publishes proceedings of those meetings; publishes a bimonthly newsletter *;login:*; produces a quarterly technical journal, *Computing Systems*; co-publishes books with The MIT Press; serves as coordinator of an exchange of software; and distributes 4.3BSD manuals and 2.10BSD tapes. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts.

Computing Systems, published quarterly in conjunction with the University of California Press, is a refereed scholarly journal devoted to chronicling the development of advanced computing systems. It uses an aggressive review cycle providing authors with the opportunity to publish new results quickly, usually within six months of submission.

The USENIX Association intends to continue these and other projects, and in addition, the Association will focus new energies on expanding the Association's activities in the areas of outreach to universities and students, improving the technical community's visibility and stature in the computing world, and continuing to improve its conferences and workshops.

The Association was formed in 1975 and incorporated in 1980 to meet the needs of the UNIX users and system maintainers who convened periodically to discuss problems and exchange ideas concerning UNIX. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the Association, differentiated primarily by the fees paid and services provided.

For further information about membership or to order publications, contact:

USENIX Association
Suite 215
2560 Ninth Street
Berkeley, CA 94710
Telephone: 510-528-8649
Email: office@usenix.org
Fax: 510-548-5738

USENIX Supporting Members

Aerospace Corporation
AT&T Information Systems
Digital Equipment Corporation
Frame Technology Corporation
Matsushita Graphic Communication Systems, Inc.
mt Xinu
Open Software Foundation
Quality Micro Systems
Rational Corp.
Sun Microsystems, Inc.
Sybase, Inc.
UUNET Technologies, Inc.